2014

Efficient Indexing for Structured and Unstructured Data

Manish Madhukar Patil
Louisiana State University and Agricultural and Mechanical College, manish.m.patil@gmail.com

EFFICIENT INDEXING FOR STRUCTURED AND UNSTRUCTURED DATA

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Manish M. Patil
Bachelor of Engineering (B.E.), Mumbai University, 2003
December 2014

Dedicated to my parents and my wife.

Acknowledgments

As with any research, this work too would have been an impossible task without the efforts of various people, who were with me throughout. I owe my success to all these people.

I would start by extending my sincerest gratitude to my advisor Dr. Rahul Shah for believing in me and giving me the opportunity to be a part of challenging and exciting research projects. His passion and drive towards research has been a constant motivating factor through all these years. His guidance and encouragement have been crucial in shaping not only my dissertation research but also my overall development as a professional researcher. Without his immense patience and support in times of uncertainties, this journey would have been much more grueling. He has been my mentor in the truest sense. For this, and many other reasons, I will always value the personal and professional experiences with him during my graduate studies.

I owe a special debt of gratitude to Dr. Wing-Kai Hon and Dr. Jeffrey S. Vitter with whom I have collaborated on numerous papers. Their guidance has been truly invaluable and I feel privileged to have the opportunity to work with them. I am grateful to Dr. Seung-Jong Park and Dr. Jianhua Chen for being a part of my thesis committee. Their willingness to evaluate my work is highly appreciated.

I am very thankful to Amy Rambhia for making me aware of the wonderful research opportunity at LSU. I would like to thank Avani Rambhia for proofreading my papers on numerous occasions and discussions there-after that had significant impact on my writing skills. I thank all my friends in the computer science department for their help and understanding throughout this PhD. A special thanks to my friend and a valued colleague Sharma Thankachan for his time to time thoughtful inputs that have made my work qualitatively better. I would also like to thank Ajay Panyala, Praveen Kondikoppa, and Vinay Amatya. Their willingness to go out of their way to use their skills for others cannot be thanked enough. Additionally, I thank all my friends in Baton Rouge, in particular Akanksha Kanitkar for lending me an ear from time to time.

I cannot go without mentioning my family. My parents, Mrs. Leela Patil and Mr. Madhukar Patil have been a source of great inspiration for me. I truly appreciate them passing their values and beliefs in education on to me. I would be remiss to not mention my wife, Amita Kasar, for everything she has had to endure over the past several years with us both continuing our educations in different continents. She has always been there for me, patiently, despite her hectic and stressful schedule and different time zones. I hope I can show her the same level of encouragement as she continues to pursue her career.

Table of Contents

List of Tables

List of Figures

Abstract

The collection of digital data is growing at an exponential rate. Data originates from wide range of data sources such as text feeds, biological sequencers, internet traffic over routers, through sensors and many other sources. To mine intelligent information from these sources, users have to query the data. Indexing techniques aim to reduce the query time by preprocessing the data. Diversity of data sources in real world makes it imperative to develop application specific indexing solutions based on the data to be queried. Data can be structured i.e., relational tables or unstructured i.e., free text. Moreover, increasingly many applications need to seamlessly analyze both kinds of data making data integration a central issue. Integrating text with structured data needs to account for missing values, errors in the data etc. Probabilistic models have been proposed recently for this purpose. These models are also useful for applications where uncertainty is inherent in data e.g. sensor networks. This dissertation aims to propose efficient indexing solutions for several problems that lie at the intersection of database and information retrieval such as joining ranked inputs, full-text documents searching etc. Other well-known problems of ranked retrieval and pattern matching are also studied under probabilistic settings. For each problem, the worst-case theoretical bounds of the proposed solutions are established and/or their practicality is demonstrated by thorough experimentation.

Chapter 1
Introduction

The world is drowning in data! There is an enormous amount of data being generated at unprecedented rates. Data emerges from text feeds, biological sequencers, internet traffic over routers, through sensors and several other sources. Due to the large volume of data, ability to query a particular dataset for mining useful/relevant information is of utmost importance. As the size of a data collection grows, the cost of executing queries over the data also increases. One of the most effective, and ubiquitous, tools for reducing query execution time is indexing. An index is a data structure that can significantly reduce the amount of data that needs to be processed when a query is executed. However, the heterogeneous nature of the data, makes it infeasible to have uniform indexing solutions across different data sources. Based on its characteristics, data can be designated as either structured or unstructured data. The term structured implies that the data is identifiable as it is organized in a structure. The most common form of structured data is a relational database table. The term unstructured data refers to any data that does not have a pre-defined structure. For example, images, videos, and text are all considered to be unstructured data.

Databases (DB) and information retrieval (IR) have evolved as separate fields primary dealing with structured and unstructured data respectively. In this dissertation, we focus on structured data in the form of relational database and unstructured data in the form of text. There are fundamental differences in the way we query a relational database and a collection of text documents as well as the properties that we expect query results to satisfy. Database systems support a structured query whereas a query to the document collection is typically free text. Knowledge of underlying data organization and its semantics (data relationships) can be exploited while indexing a database whereas documents in a collection are typically considered to be independent of each other. Moreover a bag-of-words model in information retrieval do not attach any semantics to document contents. In terms of query outputs, database systems produce exact results which are expected to satisfy

soundness and completeness properties, whereas for text documents relevance of query result is of prime importance. Thus, indexing these two types of data pose different challenges.

Traditionally, data (structured as well as unstructured) has been modeled in terms of precise values. However, recent years have witnessed increasing attention devoted to managing uncertain data due to large number of applications where uncertainty or imprecision in values is either inherent or desirable. Examples of such applications include sensor networks, data cleaning, data integration, and moving objects tracking, to name just a few. Consider a sample data cleaning application using automated methods to correct errors in data. Often, in such scenarios there is more than one reasonable alternative for the corrected value. In the standard data model, one is forced to pick one of these alternatives, which may lead to incorrectness. An uncertain model can allow multiple choices for an attribute value to be retained. With varied nature of uncertainty in data indexing solutions for precise data are often not directly portable to uncertain data. Even in situations where solutions can be ported, it is often possible to build more effective indexes for uncertain data.

In this dissertation, we propose efficient indexing solutions for a series of database and information retrieval problems, each dealing with a specific type of data. A common theme among these problems is to retrieve the few most relevant data objects instead of swamping the end-user with all data objects satisfying the query. Below, we first elaborate on such top-k query processing. We then review the uncertain data models and computational models used to capture data fuzziness and to analyze running time (efficiency) of indexing solutions respectively. Finally, remainder of this chapter gives overview of the subsequent chapters by defining the problem under consideration and outlining main contributions for each of them.

1.1 Top-k Query Processing

Database as well as information retrieval systems allow users to rank query answers. Such ranking is typically based on some scoring function. The data object score acts as a valuation for that object according to its characteristics. For example, price, year of manufacturing, number of miles driven, etc of car objects in a automobile database, or number of occurrence of query pattern in a given document. Data objects can be evaluated by a single attribute or multiple attributes that

contribute to the total object score. Thus, ranking enables access to the query answers in the order of their relevance. In many application domains, end-users are more interested in the most important (top-k) query answers in the potentially much larger answer space. Consider a user interested in ten least expensive cars manufactured after year 2010 with less than 50,000 miles on the odometer or a reader interested in a chapter that refers to the character Lily Potter the most in the Harry Potter book series. In such scenarios, our goal is to report k data objects with the highest score by employing top-k query processing. One way to answer a given top-k query is to first obtain list of data objects satisfying the input query, compute the score of each object according to scoring function, sort the objects based on their score, and return the first k objects as results. Clearly this approach is not scalable with respect to the data size. The main problem with such sort-based approach is that sorting is a bottleneck operation that requires all data objects satisfying the query to be retrieved i.e., application of query predicates is separate from ranking of query outputs. Integrating rank-awareness in query processing techniques is likely to provide a more efficient and scalable solution. By avoiding enumeration of all query outputs, such an integrated approach can achieve the query time proportional to k instead of data size or the number of query outputs and is one of the key objectives of various problems investigated in this dissertation.

1.2 Modeling Uncertainty in Data

There are two main approaches for modeling uncertain (probabilistic) relational data [133, 31]. One approach (tuple uncertainty) is to attach a probability value with each tuple - the probability captures the likelihood of the given tuple being present in the given relation. The second approach (attribute uncertainty) allows probability values at the attribute level. In this approach, a given tuple may have multiple alternatives for a given attribute. Table 1.1 shows uncertainty information expressed using tuple uncertainty. The tuples for Car id = Car1 are grouped together in a x-tuple, so they are mutually exclusive. Thus, Car1 has problems with either brakes or transmission with probability 0.1 and 0.9 respectively. Table 1.2 shows the uncertain data presented in Table 1.1 expressed using attribute uncertainty. Analogous to the models of uncertain database, two models - string-level and character-level - have been proposed recently by Jeffrey Jestes et al. [77] for

uncertain strings. A natural way of modeling an uncertain string is the string-level uncertainty model, in which all possible instances for the uncertain string are explicitly listed and they form a probability distribution function (pdf). In contrast, the character-level model describes distributions over all characters in the alphabet for each uncertain character position in the string. The character-level model is both realistic and concise in representing the uncertainty in long text strings. An uncertain string S represented in string-level model in Table 1.3, is represented in character-level model in Table 1.4.

TABLE 1.1. Example of a relation with x-tuples

Car id	Problem	Probability
Car1	Break	0.1
Car1	Tires	0.9
Car2	Transmission	0.2
Car2	Suspension	0.8

TABLE 1.2. Example of a relation with attribute uncertainty

Car id	Problem
Car1	(Break, 0.1), (Tires, 0.9)
Car2	(Transmission, 0.2), (Suspension, 0.8)

1.3 Computation Models

The first model we consider is the random access machine (RAM) model, which is probably the most popular computation model for analyzing the performance of algorithms in computer science. In this model, a computer is equipped with a CPU and memory of an unbounded size. It costs a unit of time to perform arithmetic calculation (e.g., addition, subtraction, multiplication, division), compare two numbers, read/write a word in memory, etc. (see [120] for a complete list of operations). The time complexity of an algorithm is measured in the number of operations executed; the space consumption of a data structure is measured in the number of words occupied in memory. We assume the size of a word to be $\Theta(\log n)$ bits, where n denotes the size of the problem in hand.

TABLE 1.3. String S represented in string-level model

$S =$	$\{(AAC, 0.04), (AAT, 0.06), (GAC, 0.36), (GAT, 0.54)\}$

TABLE 1.4. String S represented in character-level model

$S =$	$S[1]$	$S[2]$	$S[3]$
	(A,0.1) (G,0.9)	(A,1)	(C,0.4) (T,0.6)

When the dataset cannot be accommodated in internal memory, an algorithm typically needs to perform disk access. In this case, its running time is often dominated by its I/O cost, rather than the CPU overhead. For such a scenario, external memory model (EM) was introduced by Aggarwal and Vitter [2]. In EM, the CPU is connected directly to an internal memory, which is then connected to a much slower disk. The disk is of an unbounded size and is formatted into disjoint blocks, each of which contains B consecutive words. An I/O operation reads a block of data from the disk into memory, or conversely, writes a block of memory information into the disk. Main memory can accommodate M words and is assumed to have at least two blocks, i.e., $M \geq 2B$. The time complexity of an algorithm is measured in the number of I/Os performed; the space consumption of a structure is measured in the number of disk blocks it occupies.

1.4 Our Contributions

We deal with the indexing of structured data in Chapter 3 and 6 while we investigate problems concerning (unstructured) text data in Chapters 4, 5, and 7 as shown in Figure 1.1. Throughout the dissertation space-time complexities specified are for RAM model unless explicitly stated otherwise.

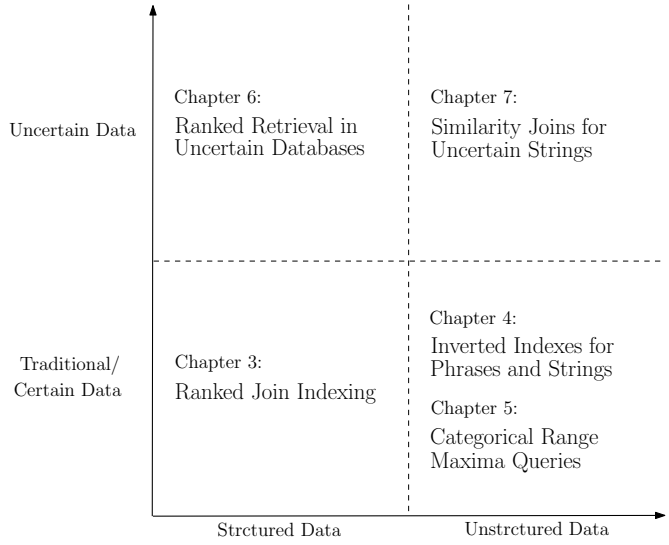

FIGURE 1.1. Dissertation overview

Chapter 3 (based on [111]): With data organized into relational tables, it is a common user requirement to correlate multiple relations for query processing through join operations. The end-user is also typically interested only in the "best" tuples which match the query. The ranked joins problem combines these two aspects and is the focus of this chapter. In top-k ranked joins, we have two input relations where each tuple has a score. The relations are joined according to joining criteria and the score of the combined tuple is monotonic function of input scores. By accessing the tuples from the relations in the ranked order, one hopes that for finding only top-k tuples, one does not have to scan through the entire relations. With the goal of avoiding unnecessary accesses to input relations, a lot of research effort has been devoted to developing stopping criteria that prunes the scanning in each relation. However, these heuristics heavily rely on scores as well as the correlation of scores between two relations. It is known that for uniformly random scores between two relations of length n, scan depth of $O(\sqrt{kn})$ is required. However, in the worst-case scenario if two relations are opposingly ranked then one might need scan depth of $(n + k)/2$. In such cases, rather than relying on scanning, it helps to preprocess the data in anticipation of such queries. We build a linear space index which explicitly writes subset of answers and calculates the rest on the fly. Based on this, we show that even if the relations are anti-correlated, one can achieve $\tilde{O}(\sqrt{kn})$ join trials to extract top-k join tuples. The experimental evaluation compares proposed indexing techniques against state-of-the-art algorithmic solution and shows superior performance.

Chapter 4 (based on [112]): This chapter considers the full-text documents searching problem. Let $\mathcal{D}= \{d_1, d_2, ..., d_{|\mathcal{D}|}\}$ be the collection of $|\mathcal{D}|$ documents of total length n. The top-k document retrieval problem is to maintain $\mathcal{D}$ as a data structure, such that given a query $Q = (P, k)$, we can report k documents with the highest $score(P, d_r)$ values. Here, $score(P, d_r)$ is any function which is dependent only on the set of occurrences of P in d_r. Inverted indexes are widely used in information retrieval for this purpose. However, the index has a shortcoming, in that only predefined pattern queries can be supported efficiently. In terms of documents where word boundaries are undefined, if we were to index all the substrings of a given document, then the storage quickly becomes quadratic in the data size. Also, if we want to apply the same type of indexes for querying

phrases or sequence of words, then the inverted index will end up storing redundant information. We present a set of inverted indexes which work naturally for strings as well as phrase searching and evaluate space-time tradeoffs for them. Techniques from succinct data structures are deployed to achieve compression while allowing fast access in terms of $score$ and document-id based retrieval. For phrase searching, we show that our indexes compare favorably against a typical inverted index deploying position-wise intersections.

Chapter 5 (based on [113]): Given an array $A[1...n]$ of n distinct elements from the set $\{1, 2, ..., n\}$, a range maximum query RMQ(a, b), returns the highest element in $A[a...b]$ along with its position. In this chapter, we study a generalization of this classical problem called *Categorical Range Maxima Query* (CRMQ) problem, in which each element $A[i]$ in the array has an associated category (color) given by $C[i] \in [\sigma]$. A query then asks to report each distinct color c appearing in $C[a...b]$ along with the highest element (and its position) in $A[a...b]$ with color c. Let p_c denote the *position* of the highest element in $A[a...b]$ with color c. We investigate two variants of this problem: a threshold version and a top-k version. In threshold version, we only need to output the colors with $A[p_c]$ more than the input threshold τ, whereas top-k variant asks for k colors with the highest $A[p_c]$ values.

In the word RAM model, we achieve linear space structure along with $O(k)$ query time, that can report colors in sorted order of $A[\cdot]$. In external memory, we present a data structure that answers queries in optimal $O(1 + \frac{k}{B})$ I/O's using almost-linear $O(n \log^* n)$ space, as well as a linear space data structure with $O(\log^* n + \frac{k}{B})$ query I/Os. Here k represents the output size, $\log^* n$ is the iterated logarithm of n and B is the block size. Further, we show that CRMQ enables us to obtain I/O-efficient data structure for top-k document retrieval problem studied in previous chapter.

Chapter 6 (based on [110]): This chapter studies the problem of ranked retrieval over uncertain databases. In traditional databases, a user defined score function assigns a score value to each tuple and a top-k query returns k tuples with the highest score. In uncertain database, top-k answer depends not only on the scores but also on the membership probabilities of tuples. Several top-k definitions covering different aspects of score-probability interplay have been proposed in the past. Most of the existing work in this research field is focused on developing efficient algorithms

for answering top-k queries on static uncertain data. Any change (insertion/deletion of a tuple or change in membership probability/score of a tuple) in underlying data forces re-computation of query answers. Such re-computations are not practical considering the dynamic nature of data in many applications. We propose a truly dynamic data structure that uses ranking function $PRF^e(\alpha)$ proposed by Li et al. [90] under the generally adopted model of x-relations [133]. PRF^e can effectively approximate various other top-k definitions on uncertain data based on the value of parameter α. For an uncertain relation with n tuples, our structure can answer top-k queries in $O(k \log n)$ time, can handle an update in $O(\log n)$ time and takes $O(n)$ space. Finally, we evaluate practical efficiency of our structure on both synthetic and real data.

Chapter 7 (based on [109]): A string similarity join finds all similar string pairs between two input string collections. It is an essential operation in many applications, such as data integration and cleaning, and has been extensively studied for deterministic strings. Increasingly, many applications have to deal with imprecise strings or strings with fuzzy information in them. This chapter presents the solution for answering similarity join queries over uncertain strings that implements possible-world semantics, using the edit distance as the measure of similarity. Given two collections of uncertain strings $\mathcal{R}$, $\mathcal{S}$, and input (k, τ), our task is to find string pairs (R, S) between collections such that $Pr(ed(R, S) \leq k) > \tau$ i.e., probability of edit distance between R and S being at most k is more than probability threshold τ. We can address the join problem by obtaining all strings in $\mathcal{S}$ that are similar to each string R in $\mathcal{R}$. However, existing solutions for answering such similarity search queries on uncertain string databases only support deterministic string as input. Exploiting these solutions would require all (exponential) possible instances of R to be considered which is not only ineffective but also prohibitively expensive. We propose various filtering techniques that give upper and (or) lower bound on $Pr(ed(R, S) \leq k)$ without enumerating possible instances for either of the strings. We then incorporate these techniques into an indexing scheme and significantly reduce the filtering overhead. Further, we alleviate the verification cost of a string pair that survives pruning by using a trie structure. Finally, effectiveness of the proposed approach is evaluated by thorough experimentation.

Chapter 2
Preliminaries

In this chapter, we briefly describe various known data structures that form the building blocks of our newly introduced indexes.

2.1 Ordered Range Retrieval

Let $A[1...n]$ be an array of score values of length n. Given a set of t non-overlapping ranges $[l_1, r_1], [l_2, r_2], ..., [l_t, r_t]$, ordered range retrieval (ORR) problem seeks k largest scores in $A[l_i, r_i]$ for $1 \leq i \leq t$ in non-increasing order. In its most simplest form query consists of a single range $[l, r]$ i.e., $t = 1$. This problem can be considered as a generalization of range maximum query (RMQ). The RMQ index is a linear-space data structure which can return the position and the value of the maximum element in any subrange $A[l...r]$ such that $0 \leq l \leq r \leq n$. Although solving RMQ can be dated back from Chazelle's original paper on range searching [28], many simplifications [14] and improvements have been made since then, culminating in Fischer et al.'s $2n + o(n)$ bit data structure [45, 46]. All these schemes can answer RMQ in $O(1)$ time. We shall use RMQ data structure to answer the ORR query. The basic result is captured in the following lemma [66].

Lemma 2.1. Let A be an array of numbers. We can preprocess A in linear time and associate A with a linear-space RMQ data structure such that given a set of t non-overlapping ranges $[l_1, r_1], [l_2, r_2], ..., [l_t, r_t]$, we can find the k highest scoring entries in non-increasing order of score in $A[l_1, r_1] \cup A[l_2, r_2] \cup ... \cup A[l_t, r_t]$ in $O(t + k \log k)$ time.

2.2 Restricted Ordered Range Retrieval

Let $A[1...n]$ be an array where each entry is associated with three values $select$, $join$, $score$. Restricted ordered range retrieval (RORR) seeks k highest scoring entries along with their scores among those entries $A[i]$ which satisfy input constraints. We consider following two variants of this problem which differ in constraints that the $join$ values of the array A can be subjected to.

Lemma 2.2. Let $A[1...n]$ be an array where each entry is a triplet of the form ($select$, $join$, $score$). We can associate A with a $O(n)$ space data structure, such that given a range $[s_l, s_r]$ and parameters j_e, k, we can search among those entries $A[i]$ with $s_l \leq A[i].select \leq s_r$, $A[i].join = j_e$, and report the k highest scoring entries in non-increasing order of score by spending $O(\log k)$ time per answer after initial query set up cost of $O(\log n)$.

Proof. Let $A^{j,s}$ denote a list of entries from an array A such that they are first sorted based on $join$ values and ties are broken by ordering based on the $select$ values. We maintain such a list $A^{j,s}$ along with a RMQ structure on the $score$ values associated with entries in the list $A^{j,s}$ as an index. To answer the query, we begin by performing a binary search to obtain the boundary $[l, r]$ in $A^{j,s}$ such that all the entries in the subrange $A^{j,s}[l...r]$ qualify the given constraints i.e., $A^{j,s}[i].join = j_e$ and $s_l \leq A^{j,s}[i].select \leq s_r$ for $l \leq i \leq r$. Now RMQ component can be used to retrieve array entries in the non-increasing order of the $score$ values as described in Lemma 2.1. □

Lemma 2.3. Let $A[1...n]$ be an array where each entry is a triplet of the form ($select$, $join$, $score$). We can associate A with a $O(n)$ space data structure, such that given two ranges $[s_l, s_r]$, $[j_l, j_r]$, and a parameter k, we can search among those entries $A[i]$ with $s_l \leq A[i].select \leq s_r$, $j_l \leq A[i].join \leq j_r$, and report the k highest scoring entries in non-increasing order of score by spending $O(\log n)$ time per answer after initial query set up cost of $O(\log^2 n)$.

Proof. The above RORR query can be answered by directly using the result from [65]. However, this approach returns k answers in an unsorted order. In order to get faster query time, authors use a variant of the Lemma 2.1 in their algorithm by allowing the answers to be unsorted. Since we need to retrieve top-k answers by paying the cost on per-answer basis, only change required in the solution proposed in [65], is to use Lemma 2.1 instead of its variant. □

2.3 Three-dimensional Dominance Reporting

Given a set S of n points in three dimensions and a query point $q = (q_1, q_2, q_3)$, the three-dimensional dominance reporting asks for all the points $s = (x_1, x_2, x_3) \in S$ such that $x_i < q_i$, $1 \leq i \leq 3$. Vengroff and Vitter [131] addressed this problem in the external memory model and

proposed an $O(n \log n)$-space data structure that can answer queries in optimal $O(\log_B n + k/B)$ I/Os. The best known result for the problem is by Afshani [1] which achieves linear space along with same optimal I/O bound.

2.4 Three-sided Orthogonal Range Reporting

Given a set S of n points in two dimensions, three-sided orthogonal range reporting asks for all points inside a query rectangle of the form $[x_1, x_2] \times (-\infty, y]$. The best EM model solution to the two-dimensional three-sided range reporting problem is due to Arge et al. [8] which takes linear space and reports all the points inside the query rectangle in $O(\log_B n + k/B)$ I/Os. When the two-dimensional points are on the $[n] \times [n]$ grid, Larsen et. al [85] achieve improved query bound of $O(1 + k/B)$ I/Os.

2.5 Suffix Trees and Compressed Suffix Trees

Given a text $T[1...n]$, a substring $T[i...n]$ with $1 \leq i \leq n$ is called a suffix of T. The lexicographic arrangement of all n suffixes of T in a compact trie is known as the *suffix tree* of T [132], where the ith leftmost leaf represents the ith lexicographically smallest suffix. Each edge in the suffix tree is labeled by a character string and for any node u, $path(u)$ is the string formed by concatenating the edge labels from root to u. For any leaf v, $path(v)$ is exactly the suffix corresponding to v. For a given pattern P, a node u is defined as the *locus node* of P if it is the node closest to the root such that P is a prefix of $path(u)$; such a node can be determined in $O(p)$ time, where p denotes the length of P. The *generalized suffix tree* (GST) is a compact trie which stores all suffixes of all strings in a given collection $\mathcal{D}$ of strings. The drawback of the suffix tree is its huge space consumption, which requires $O(n \log n)$ bits in theory. Yet, it can perform pattern matching in optimal $O(p + |output|)$ time, where $|output|$ is the number of occurrences of P in T. Compressed suffix tree (CST) is a space-efficient version of suffix tree. Several variants of CSTs have been proposed to date [98, 54, 118, 117, 47, 107, 130, 21, 106]. String B-tree (SBT) [43] for a text T can be thought of as an external memory counterpart of suffix tree as it occupies $\Theta(n/B)$ blocks or $\Theta(n \log n)$ bits space and can locate the locus node of pattern P in $O(p/B + \log_B n)$ I/Os.

2.6 Bit Vectors with Rank/Select Support

Let $\mathcal{B}[1..n]$ be a bit vector with its m bits set to 1. Then, $rank_{\mathcal{B}}(i)$ represents the number of 1's in $\mathcal{B}[1..i]$ and $select_{\mathcal{B}}(j)$ represents the position in $\mathcal{B}$ where the jth 1 occurs (if $j > m$, return NIL). There exists representations of $\mathcal{B}$ in $n + o(n)$ bits and $m \log(n/m) + O(m) + o(n)$ bits, which can support both $rank_{\mathcal{B}}(\cdot)$ and $select_{\mathcal{B}}(\cdot)$ operations in constant time. These structures are known as fully indexible dictionaries. Another representation, where the space occupancy is $m \log(n/m) + O(m)$ bit support only $select_{\mathcal{B}}(\cdot)$ operation in constant time and is known as indexible dictionary [116].

Chapter 3
Ranked Join Indexing

3.1 Introduction

Ranking queries are useful in focusing attention on the most important answers to a query from larger answer space. In top-k join queries, a "join" condition among tuples in different input relations joins them together in one output join result. Each join result has a combined score computed from the scores of participating tuples. The goal is to produce the top-k join results based on the combined score. Thus, top-k join query is essentially a multi-criteria optimization query that combines the individual scores into one global score by applying the provided aggregation function. Real-life examples of multi-criteria optimization, are given below.

Example 1. A family is interested in buying a 3 bedroom house with a school nearby having at least 500 students, with the objective of minimizing the total cost. Consider a simple cost function that sums the price of the house and 5-year school tuition. Searching the two web databases, HOUSES and SCHOOLS, the family issues the following query:

```
SELECT * FROM HOUSES H, SCHOOLS S
WHERE H.location = S.location AND H.no_of_bedrooms = 3
AND S.no_of_students >= 500
ORDER BY H.price + 5 * S.tuition LIMIT 10
```

Example 2. A tourist is looking for a good restaurant to have dinner. A local information website can provide the list of restaurants along with information about their locations and cost (average price for a diner). Also restaurant ratings are typically available through websites (such as Zagat-Review), where food rating for a restaurant is given by a number between 1 and 30. By imposing some constraints on dinner cost and restaurant rating, the tourist can issue the following query:

```
SELECT * FROM RESTAURANTS R, REVIEWS S
WHERE R.id = S.rest_id AND 20 <= R.price <= 45 AND S.rating > 10
ORDER BY S.rating/R.price DESC LIMIT 10
```

Such top-k join queries can be answered in a naive way as follows: First, the input relations are filtered to retrieve tuples satisfying the given predicates, which are then joined according to the join condition. For each join result, the global score is computed according to the given scoring function. Finally, the results are sorted on the computed combined score to produce the top-k results. With the goal of being more efficient than the naive approach, several algorithms have been proposed till date for answering top-k join queries [100, 25, 73, 93, 3]. Most of these algorithms take relations filtered based on given predicates as input lists. These lists typically support sorted access i.e., tuples can be retrieved in a ranked order as determined by their scores. Algorithm proceeds by incrementally retrieving the tuples from the input lists and maintain aggregate score of all "seen" tuples. Algorithm terminates when "enough" information to decide on the top ranked join results are obtained. This stopping mechanism determines the number of tuples accessed from the input lists (scan depth) for answering a query.

Primary focus of the work on top-k join processing so far has been to derive tighter early termination conditions while navigating the cartesian product of input lists systematically. Unfortunately, effectiveness of such "stopping mechanism" heavily depends on the correlation between the input lists. Consider the sample database shown in Table 3.1 for the example queries described earlier. Let us assume we are interested only in the top-1 result for both examples. It can be seen that for a tourist to decide the best possible choice, almost all the restaurants need to be evaluated whereas a family can decide on the best choice by evaluating three houses and three schools only. The main factor that can cause worst case scenario, as evident in the tourist example, is the "curse of

TABLE 3.1. Houses (H) and Schools (S) database

(a) HOUSES

id	location	price (in K)
1	3	39
2	**1**	**40**
3	1	42
4	2	45
5	3	45
6	2	46
7	3	48
8	3	48
9	3	50

(b) SCHOOLS

id	location	tuition (in K)
1	2	2.4
2	**1**	**2.5**
3	1	2.8
4	3	3.0
5	2	3.2
6	3	3.3
7	3	3.3

anti-correlation". A restaurant which costs less (ranked higher in terms of affordability) typically has lower ratings. Whereas input data for the example 1 is positively correlated as a good locality typically has better schools with higher tuition fees and houses with higher costs. While most of the algorithms proposed are efficient when the input lists to be joined are positively correlated, they need to access sizable amount of the lists leading to poor performance otherwise.

It is known that [74], even for the input lists of length n with uniformly random scores, scan depth of $O(\sqrt{kn})$ is required. However, in the worst-case scenario if two lists are anti-correlated then one might need scan depth of $O(k+n)$. Motivated by the limitations of the algorithmic approach, we take an indexing approach for answering top-k join queries. We note that top-k join query processing is closely related with other fundamental problems in database community such as top-k selection queries and skyline computation. However, despite their similarities, extending/adopting the indexing solutions of these problems to support top-k join queries is challenging.

Top-k selection queries: For top-k select queries, all input lists contain the same set of objects ranked on different criteria i.e., all the objects can be thought to be a part of a single relation, where each object has a set of score attributes and the goal is to select the best k objects according to some combination (aggregation) of these score attributes. Thus, a top-k selection query can be regarded as a special case of a top-k join query when there is a one-one mapping among the tuples in relations involved in the join query as in Example 2.

Computing skyline: Given a set of multi-dimensional objects, skyline queries find the set of interesting (i.e., non-dominated) objects. A m-dimensional object P dominates another object Q if P is better than or equal to Q in all m dimensions, and strictly better than Q in at least one dimension. For the first example query described above, consider all (house, school) pairs obtained by joining relation HOUSES (H) with SCHOOLS (S) which satisfies number of bedrooms and number of students in school criteria. When these points are plotted in two-dimensional plane with H.price as its x coordinate and 5 * S.tuition as its y coordinate, any point P in the skyline will have its combined score better (lower) than any point Q not in the skyline. Therefore a top-k join query can be thought of as selecting top-k skyline points.

We propose a data structure for efficient top-k join processing in this chapter. The proposed index explicitly writes subset of answers so as to reduce the number of tuples that need to be accessed during query time. Our index achieves the goal of performing at most $\tilde{O}(\sqrt{kn})$[1] join trials to extract top-k joined tuples, even if the input lists are anti-correlated while occupying the space linear to the input data size. The proposed data structure also integrates evaluation of predicates on the relations involved in join with query processing which is external to most of the existing solutions. Our extensive experimental study under different parameter settings shows that our index yield high performance gains against the well know Rank-Join algorithm by Ilyas et al. [73].

3.2 Problem Statement

Given a set of relations R_1 to R_γ such that each relation R_i is associated with a set of attributes $C_i = \{\alpha_i^1, \alpha_i^2, ..., \alpha_i^c\}$ and a ranking function, which assigns a score to every tuple $t \in R_i$ denoted by $score_i(t)$, preprocess these relations and construct an index so as to answer the top-k join queries efficiently. In the query, we assume $join\text{-}condition$ associates those tuples from two relations which satisfy the corresponding $select\text{-}predicate$ (a range query on one of its attributes). Results of the join query are ranked using a monotone function F which computes the total score of a tuple by combining its scores in individual relations. Finally, let `LIMIT` controls the number of results reported to the user. Without loss of generality, now onwards we assume higher value of $score_i$ is preferred and F is a monotonic non-decreasing function i.e., we would like to retrieve k join results with the highest combined score computed using function F. Also let each relation R_i has n tuples and $N = \gamma cn$ is the total size of all γ relations.

3.3 Achieving Worst Case Query Time of $\tilde{O}(\sqrt{kn})$

This section describes the proposed linear space index which can answer top-k join queries involving two relations in $\tilde{O}(\sqrt{kn})$ time. Without loss of generality, let the $select\text{-}predicate$ specifies a range query on attribute α_i^s and join is to be performed on attribute α_i^j. We first explain the simpler version of the index for the case where join operation is restricted to equality join. We

[1]The notation $\tilde{O}$ ignores poly-logarithmic factors.

also assume that (1) relations involved in the top-k join query i.e., R_1, R_2 and (2) join, selection attributes i.e., α_1^j, α_2^j, α_1^s, α_2^s are predefined to begin with. Let $L_i^s = \{t_1, t_2, ..., t_n\}$ denote a list of tuples from R_i sorted based on selection attribute i.e., α_i^s and $[l_i, r_i]$ be the range in list L_i^s obtained by applying the given $select\text{-}predicate$ on R_i. Now, our task is to join tuples $\{t_x | t_x \in L_1^s$, $l_1 \leq x \leq r_1\}$ with $\{t_y | t_y \in L_2^s, l_2 \leq y \leq r_2\}$ and retrieve the top-k highest scored pairs. A naive way of performing top-k join would result in the worst case $O(n^2 \log n)$ algorithm. Our idea is to preprocess the relations R_1, R_2 and store some partial answers so that top-k join queries can be efficiently answered without going through the entire list of tuples.

3.3.1 Index Structure

Our index consists of three components namely binary trees, score-matrices, RORR structures and are described below.

(1) Binary trees: In our index, we maintain a balanced binary tree representation of both the relations R_1 and R_2. Let Δ_i^s be the balanced binary tree (of n leaves) built over the list L_i^s i.e., each leaf in Δ_i^s corresponds to a tuple in relation R_i and leaves are sorted by selection attribute α_i^s. Evaluation of $select\text{-}predicate$ (a range query) on attribute α_i^s can now be performed by a simple binary search on Δ_i^s to obtain a range $[l_i, r_i]$ in the list L_i^s.

(2) Score-matrices: Consider a naive way of storing answers for all possible queries. The number of possible contiguous ranges in L_i^s is $\binom{n}{2} = O(n^2)$. Therefore, if we preprocess these lists and store the top-k answers for all pairs of ranges (between L_1^s and L_2^s, based on given join condition, for a fixed k), top-k join query can be answered in optimal $O(k)$ time. However, the space required for storing all answers $O(n^4 k)$ (for a fixed k) is not practical at all.

A key idea to reduce the space requirement without increasing query time too much is to store the answers between only selected pairs of ranges. Each node u of Δ_i^s naturally corresponds to a range covering all tuples represented by the leaves in the subtree rooted at node u. Let $L_i^s(u)$ denotes the list of these tuples and $g = \tilde{O}(\sqrt{kn})$ be a grouping parameter. A node u of Δ_i^s is called a heavy node if the number of leaves in its subtree is more than a parameter g, i.e $|L_i^s(u)| > g$. It

can be easily verified that the number of heavy nodes in Δ_i^s is $O(n/g)$. Our index stores the top-k answers (for a fixed k) for pairs (u, v) computed using predefined join condition, where u is a heavy node in Δ_1^s and v is a heavy node in Δ_2^s. Here final score of a tuple after join is computed using monotonic function $F(score_1, score_2)$. This information is stored in a two-dimensional matrix which we call a score-matrix of dimensions $O(n/g) \times O(n/g)$ with each entry of the matrix storing top-k answers. Since k is an online parameter and we do not want our index to be tied to a fixed value of k, we maintain score-matrix for $k = 1, 2, 4, 8, ...$ separately i.e., we maintain a collection of $\log n$ score-matrices. Though this collection of score-matrices is tied to a scoring function at the construction time, later (Section 3.3.7) we describe how such a collection can be used to answer a range of monotonic linear scoring functions.

(3) RORR structures: As explained earlier, score-matrix stores the top-k answers for selected pairs of ranges. For top-k queries that can not be answered using score-matrix only, we need to do some on-the-fly computations. RORR structure is intended to accelerate these computations during query execution. We maintain a RORR index (Lemma 2.2) for the relation R_i such that values of the triplets ($select$, $join$, $score$) of an array on which RORR structure is to be built are populated using attributes α_i^s, α_i^j and scoring function $score_i$ respectively.

Before we move on to the query algorithm, we highlight the important properties of our structure.

Lemma 3.1. Given any contiguous range $[l, r]$ in a list L_i^s, it can be divided into $h < 2 \log n$ subranges such that, each of this subrange corresponds to the list $L_i^s(u)$, where u is a node in Δ_i^s.

Proof. Follows from the properties of a balanced binary tree. □

Using Lemma 3.1 and the condition for a node to be heavy in balanced binary tree Δ_i^s, we write the following lemma.

Lemma 3.2. Given any contiguous range $[l, r]$ in a list L_i^s, it can be divided into 3 subranges: $[l, l' - 1], [l', r']$ and $[r' + 1, r]$, such that $l' - l < g, r - r' < g$ and the subrange $[l', r']$ can be further divided into $h < 2 \log n$ sub-subranges such that each of this sub-subrange is of the form $L_i^s(u)$, u being a heavy node in Δ_i^s.

Proof. For simplicity, assume n and g to be a power of 2 then any range $[mg, (m+1)g-1]$ for $m \geq 0$ will be of the form $L_i^s(u)$, u being a heavy node in Δ_i^s. Therefore the range $[l, r]$ can be divided into 3 subranges: $[l, l'-1], [l', r']$ and $[r'+1, r]$ such that $l' = g\lceil l/g \rceil - 1$ and $r' = g\lfloor r/g \rfloor$. □

3.3.2 Query Algorithm

To answer the top-k join query, our query algorithm sequentially executes following steps:

(1) Query binary trees: Query algorithm begins with filtering of tuples in the relations R_1 and R_2 based on given $select\text{-}predicates$. Since the $select\text{-}predi\text{-}\ cate$ is a range query, we can obtain a contiguous range $[l_i, r_i]$ in the list L_i^s using binary tree Δ_i^s such that each tuple $t_x \in R_i$, $l_i \leq x \leq r_i$ satisfies the given predicate. In order to reduce the computations performed during query execution, we would like to use the pre-computed top-k answers from the score-matrix. To enable such a lookup we divide the range $[l_i, r_i]$ using Lemma 3.2 into three subranges $[l_i, l_i'-1], [l_i', r_i']$ and $[r_i'+1, r_i]$. Now, we can split the main task of answering top-k join query between the ranges $[l_1, r_1]$ and $[l_2, r_2]$ into the subtasks of answering top-k join queries between following subranges.

(1) $[l_1', r_1']$ and $[l_2', r_2']$

(2) $[l_1, l_1'-1]$ and $[l_2, r_2]$

(3) $[r_1'+1, r_1]$ and $[l_2, r_2]$

(4) $[l_1, r_1]$ and $[l_2, l_2'-1]$

(5) $[l_1, r_1]$ and $[r_2'+1, r_2]$

Top-k answers for the first subtask are obtained by querying the appropriate score-matrix in Step 2. Whereas Step 3 of the query algorithm efficiently computes top-k answers for the remaining subtasks by querying the appropriate RORR structures. We note that the pairs $[l_1', r_1']$, $[l_2, r_2]$ and $[l_1, r_1]$, $[l_2', r_2']$ need not be considered, as these cases are subsumed by the five cases listed above.

(2) Query score-matrix: Using Lemma 3.1 and 3.2, for a range $[l_i', r_i']$ from the previous step, we can obtain a set S_i of $O(\log n)$ nodes such that: (1) each node in S_i is a heavy node in Δ_i^s, (2) the subtrees of any two nodes in S_i are disjoint and (3) the subtrees of the nodes in S_i together contain exactly all the tuples in R_i that are covered by the range $[l_i', r_i']$.

Top-k answers for the first subtask can now be retrieved by querying score-matrix component $O(\log^2 n)$ times once for each pair of nodes $(u, v) \in S_1 \times S_2$. We choose appropriate score-matrix based on the online query parameter k i.e., the one which stores top-(2^m) answers, $2^{m-1} < k \leq 2^m$. From each query (among $O(\log^2 n)$ queries), we retrieve only the top-1 answer, and put them into a max-heap (binary search tree). In each iteration, we do the following: (1) We perform the extract-max operation on the heap and add it to our answer list. (2) Let (u, v) be the pair of nodes which has contributed the answer just extracted. We query the cell of the score-matrix corresponding to the pair (u, v) to retrieve the next highest ranked answer and insert it into the heap. Thus, after k iterations, we get the top-k answers as required for the first subtask.

(3) Query RORR structure: We demonstrate the steps involved in the query algorithm by considering subtask 2 as a representative case, other subtasks can be handled in a similar way. Before we explain how to answer the top-k join query between ranges $[l_1, l'_1 - 1]$ and $[l_2, r_2]$ for subtask 2, we show how to retrieve the join results in non-increasing order of a combined score for a given tuple $t_x \in R_1$ such that $l_1 \leq x \leq l'_1 - 1$. We initiate a query to the RORR structure for relation R_2 (Lemma 2.2) with parameter j_e set to the join value of tuple t_x and range $[s_l, s_r]$ as dictated by $select\text{-}predicate(R_2)$. Now this query can be used to retrieve tuples from R_2, which satisfy the input predicate and can produce valid join combinations with t_x, in the non-increasing order of $score_2$. We note that since $score_1(t_x)$ is fixed and ranking function $F(score_1, score_2)$ is monotonic, ordering of the tuples based on $score_2$ as given by the RORR structure is same as the ordering based on combined score.

To obtain top-k join results for a pair of range ($[l_1, l'_1 - 1]$, $[l_2, r_2]$) we use the following procedure: For each tuple t_x in $[l_1, l'_1 - 1]$, we initiate the RORR query with appropriate query parameters as explained above and find the tuple t_y in $[l_2, r_2]$, that gives the maximum combined score. All these top-1 answers are inserted into a max-heap (binary search tree). Then, in each of the k iterations, we do the following: (1) We perform the extract-max operation on the heap and add it to our answer list. (2) Let t_x be the tuple in R_1 which has contributed the answer just extracted. We use the RORR query initiated for t_x to retrieve the next highest ranked join result for t_x and insert it into the heap.

(4) Top-k reporting: This step of the query algorithm simply combines the top-k answers obtained in previous steps for each of subtasks to obtain the top-k answers for the main task of joining the tuples $\{t_x | t_x \in L_1^s, l_1 \leq x \leq r_1\}$ with $\{t_y | t_y \in L_2^s, l_2 \leq y \leq r_2\}$. This step can be executed in time linear to input parameter k.

For the subtasks 2 to 5, one of the two ranges will be small ($\leq g$). We call the tuples belonging to these small ranges as fringe tuples. A tuple pair appearing in the final top-k answers will either have at least one fringe tuple or both of its tuples can be non-fringe. Subtasks 2 to 5 cover the former scenario whereas tuple pair with both non-fringe tuples will be returned as an answer during execution of the subtask 1. Thus all top-k answers will be found by our query algorithm. However, subtask division presented above leads to duplicate join results being reported. This can be avoided by simply replacing the subtasks as (4) $[l_2, l_2' - 1]$ and $[l_1', r_1']$, (5) $[r_2' + 1, r_2]$ and $[l_1', r_1']$ without affecting the correctness of algorithm.

Figure 3.1 shows the overview of the query algorithm when applied to Example 2. Restaurants which satisfy the input predicates i.e., tuples in sample database of Table 3.1, are shown as shaded leaves. Since $n = 16$ in this case, to retrieve the top-1 answer we choose grouping parameter $g = 8$. Best choice for the tourist can now be decided by on-the-fly evaluation of 3 joins using RORR structures in addition to a score-matrix lookup.

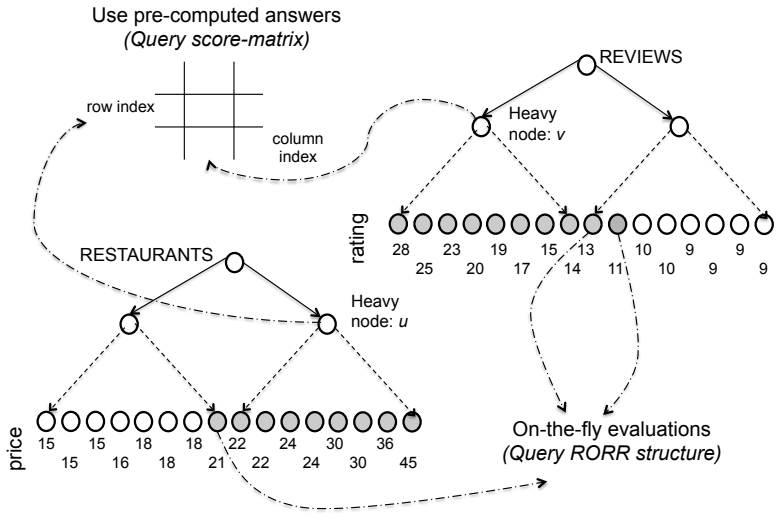

FIGURE 3.1. Overview of query algorithm

3.3.3 Space-Time Analysis

This subsection analyzes the performance of our structure. We will also fix the value of grouping parameter g to strike a good balance between space and query time. We begin by bounding the space and query complexities as mentioned in the following lemma.

Lemma 3.3. Our structure uses $O(n + n^2k \log n/g^2)$ space, and answers top-k join query in $O(\log^3 n) + O((g + k) \log n)$ time.

Proof. For a relation R_i, we maintain a balanced binary tree representation and a RORR structure and each of these two structures occupies a linear space. Since there are only two relations, total space required for all binary trees and all RORR structures can be bounded by $O(n)$. Our structure also maintains a collection of $O(\log n)$ score-matrices where size of the each score-matrix is bounded by $O(n/g) \times O(n/g) \times k = O(n^2k/g^2)$. Therefore, the proposed index structure occupies $O(n) + O(n^2k \log n/g^2)$ space.

To obtain the query complexity, we analyze the time spent by the query algorithm in each of the four steps. Recall that the first step of algorithm simply obtains a range $[l_i, r_i]$ using binary tree Δ_i^s and splits it into subranges based on Lemma 3.2 for both the relations R_1, R_2. Thus time complexity of Step 1 can be bounded by $O(\log n)$. Time required for querying score-matrix in Step 2 of the algorithm can be bounded by $O((\log^2 n + k) \log n)$ as heap contains at most $O(\log^2 n + k)$ elements when kth highest join result for subtask 1 is retrieved. A close look at the algorithm reveals that querying RORR structure (Step 3) dominates the query cost. Again we use subtask 2 as a representative and query cost for the other subtasks can be bounded in a similar way. We split the time required for answering subtask 2 as follows: (1) For each tuple $t_x \in R_1$ such that $l_1 \leq x \leq l'_1 - 1$, we initiate a query to RORR structure on R_2 which has initial query set up cost of $O(\log n)$ before any answers can be retrieved (Lemma 2.2). Since $l'_1 - l < g$ (Lemma 3.2), total cost can be bounded by $O(g \log n)$. (2) Time required for retrieving total of $O(g + k)$ answers from a collection of RORR queries can be bounded by $O((g+k) \log k)$ (Lemma 2.2). With heap containing at most $O(g + k)$ elements when top-k answers for subtask 2 are found, cost of heap operations can be bounded by $O((g + k) \log n)$. Thus querying RORR structure takes $O((g + k) \log n)$ time.

The last step of query algorithm combines top-k answers from each of the five subtasks to produce top-k answers for the original top-k join query in $O(k)$ time. Therefore, our query algorithm can answer top-k join queries in $O(\log^3 n + (g+k)\log n)$ time. □

Depending on the choice of g, Lemma 3.3 gives various tradeoffs between space and query time. To achieve linear space, we choose $g = \sqrt{nk \log n}$, which establishes the result summarized below.

Theorem 3.4. Given relations R_1 and R_2 of size n, top-k join queries with (1) equality join on a predefined attribute and (2) both the relations being subjected to a range query on a predefined attribute, can be answered in $O(\sqrt{nk}\log^{3/2} n)$ time by maintaining an index of size $O(n)$.

3.3.4 Index Construction

From the index description, it can be seen that construction of each binary tree and RORR structure can be achieved in $O(n \log n)$ time. A naive way of populating score-matrix would result in quadratic construction time. However, we can first construct binary trees and RORR structures and then use the procedure described in Step 3 of the query algorithm to achieve $\tilde{O}(n^{3/2})$ construction time. An important observation that allows more efficient index construction is that, we do not need to explicitly compute top-k answers for each heavy node pair (u, v). Let $u_l and u_r$ be the left and right child of a node u in Δ_1^s respectively with both being heavy nodes. We observe that for a tuple pair (t_x, t_y) to be in top-k answers for (u, v), it must be in top-k answers for (u_l, v) or (u_r, v). Thus, top-k answers for (u, v) can be computed by simply scanning at most $2k$ join results.

Further, we would like to highlight that the proposed index can be made semi-dynamic (insertion only). Balanced binary trees can handle insertions efficiently whereas RORR structures summarized in Lemma 2.2 and 2.3 can be replaced by their dynamic counterparts [64, 91] still maintaining linear index space. We can easily obtain $\tilde{O}(n^{3/4})$ query time and (amortized) update time solution by keeping track of newly inserted tuples and reconstructing all the score-matrices only after $O(n^{3/4})$ tuples are inserted to a relation. However, these $O(n^{3/4})$ tuples now need to be evaluated during query time using RORR structures in the same way fringe tuples are evaluated in Step 2 of the query algorithm. We observe that $\tilde{O}(\sqrt{kn})$ query time as well as $\tilde{O}(\sqrt{n})$ update time can be obtained by

balancing the cost of evaluating fringe leaves and that of newly inserted tuples. We can achieve the same by reconstructing the score-matrix entries related to a particular heavy node only if g tuples are inserted in its subtree.

3.3.5 Supporting Arbitrary Relations, Selection and Join attributes

To support generalized top-k queries, for each relation R_i, we now maintain a balanced binary tree representation for each of its c attributes. As RORR structure depends on the selection as well as join attribute, we need to maintain RORR index for c^2 pairs of attributes for each of the γ relations. Similarly, for a given pair of relations we maintain score-matrices for all possible combinations of join and selection attributes. Furthermore, as two relations involved in a join query can be picked up in γ^2 ways, we store score-matrices for each of these combinations as well. As before, score-matrices are maintained for each $k = 1, 2, 4, 8, ...$ Finally, to limit the total space requirement of our index we choose threshold $g = \sqrt{\gamma c^2 nk \log n}$. For answering a query, we follow the same procedure as described earlier by choosing the appropriate balanced binary trees, score-matrix and RORR structure at each step of the query algorithm. By following similar analysis as used in previous subsection, Theorem 3.4 can be rewritten as follows.

Theorem 3.5. Top-k join queries involving any two relations out of γ relations with (1) equality join on a single attribute and (2) each relation being subjected to a range query on one of its attributes, can be answered in $O(\sqrt{cNk}\log^{3/2} n)$ time by maintaing an index with $O(cN)$ size and $\tilde{O}((cN)^{3/2})$ construction time, where c is the number of attributes per relation, n is the number of tuples per relation, and $N = \gamma cn$ is the total size of all γ relations.

3.3.6 Supporting Inequality Joins

To support inequality joins, instead of using the RORR structure from Lemma 2.2, we now use the one in Lemma 2.3. We note that, with this change our index can support both equality as well as inequality joins making it generic, without hurting its space requirement. Here, we only highlight the difference in the way RORR structure is used by our query algorithm, as rest of the operations remain unchanged. For illustration purpose, we again use subtask 2 and assume the join operation

to be $\alpha_1^j \leq \alpha_2^j$. Given a tuple $t_x \in R_1$ with $l_1 \leq x \leq l_1' - 1$, to retrieve the valid join combinations it produces for subtask 2, we query appropriate RORR structure for relation R_2 (based on α_2^i, α_2^j). The $select\text{-}predicate(R_2)$ determines the range $[s_l, s_r]$ and for the query parameter range $[j_l, j_r]$, we set j_l to minimum join value in relation R_1 and j_r to join value of tuple t_x. Thus, we achieve an index with space requirement as that of index in Theorem 3.5 but with query time $O(\sqrt{cNk}\log^{5/2} n)$.

3.3.7 Supporting Approximate Monotonic Linear Scoring Functions

In this subsection we show how an index built for a particular scoring function can be used to approximately answer top-k join queries with scoring functions that incorporate user preferences. For presenting the ideas, we assume the index has been built with aggregate scoring function as sum i.e., $F(score_1, score_2) = score_1 + score_2$ and all scores are positive integers. Such an index can directly support exact top-k join results for all functions $\delta(score_1 + score_2)$, where δ is a scaling factor. For scoring functions with arbitrary user preferences i.e., $(\delta score_1 + \beta score_2)$, we would like to achieve $(1+\epsilon)$ approximation for $\epsilon > 0$. Thus, our goal in this case is to retrieve top-k answers such that score of all the returned top-k answers is at least $1/(1+\epsilon)$ times that of the actual top-kth answer. To achieve this we build score-matrices with scoring functions $score_1 + score_2/(1+\epsilon)^i$ and $score_1/(1+\epsilon)^i + score_2$. Therefore, for a particular value of k we now have $O((\log m)/\epsilon)$ score-matrices as opposed to just one score-matrix used before, where $m = max(score_1, score_2)$. Space complexity of the index can be maintained to be linear despite increased number of score-matrices by adjusting the grouping parameter appropriately. For example, linear space bound can be obtained for index in Theorem 3.4 by choosing $g = \sqrt{(kn \log n \log m)/\epsilon}$ thus compromising query time only by the factor of $\sqrt{(\log m)/\epsilon}$. We note that ϵ is a query-space tradeoff parameter i.e., we can achieve the same query time as before with increased index space by a factor of $(\log m)/\epsilon$.

3.4 Adapting for Positively Correlated Data

The index proposed in the previous section has been developed to achieve the goal of providing theoretical guarantee for top-k join query. As a side-effect, the proposed index is insensitive to the correlation between two input lists to be joined. When the input lists are positively correlated, scan

depth is typically only a small fraction of their lengths. However, our index ignoring the correlation may end up accessing more number of tuples (determined by g) than the scan depth. Below we show how our index can take the advantage of positively correlated data to achieve better performance. We will first extend the index structure and then incrementally improve the query algorithm.

Index structure: In addition to the three components of our index described earlier, we maintain a component called "RMQ Structures" so as to support ordered range retrieval for any relation R_i. Recall that L_i^s denotes a list of tuples from R_i sorted based on selection attribute i.e., α_i^s. We maintain a RMQ structure on the score values associated with tuples in list L_i^s based on ranking function $score_i$. Now all tuples in R_i satisfying the given $select\text{-}predicate$ (a range query) on attribute α_i^s can be retrieved in non-increasing order of their score efficiently (Section 2.1). As there are γ relations and for each relation we have c choices for selection attribute α_i^s, total space required for RMQ structures can be bounded by $O(\gamma cn) = O(N)$. Thus space complexity of our index remains unchanged.

Query algorithm: From Section 3.3.3 we know that, Step 3 of the query algorithm i.e., querying RORR structure dominates the query cost. This step aims to return top-k answers for subtasks 2 to 5, obtained by splitting main task of answering top-k join query between the ranges $[l_1, r_1]$ and $[l_2, r_2]$. For these subtasks, one of the two ranges is small ($\leq g$) by Lemma 3.2. Query algorithm presented in previous section requires at least one join result to be computed for each tuple in the small list. When input lists are positively correlated, an immediate improvement can be obtained by using any of the heuristics proposed earlier to obtain top-k answers for each of these subtasks. Though any heuristic can be applied, here we use a well known Rank-Join algorithm proposed by Ilyas et al. [73]. Therefore, Step 3 of the query algorithm now creates four instances for Rank-Join algorithm for handling each of the subtasks 2 to 5. In each step of the algorithm, Rank-Join retrieves the next highest ranked tuple from one of two input lists. Sorted access of tuples required by Rank-Join is supported by RMQ structures component of the index. In addition to the immediate gains, use of Rank-Join offers another subtle benefit: ability to merge/overlap the subtasks as we only need collective top-k answers from all subtasks.

Merging of subtasks: A close look at the four subtasks reveals the fact that these subtasks are not entirely independent. Subtasks 2 and 3, have list of tuples in the range $[l_2, r_2]$ common between them. Use of Rank-Join allows us to merge the related subtasks, effectively reducing the combined efforts spent on the individual subtasks.

Let $L_{2,3}$ be the list obtained by merging the tuples in the range $[l_1, l'_1 - 1]$ (subtask 2) and $[r'_1 + 1, r_1]$ (subtask 3) of relation R_1. In a merged task Rank-Join can then operate on the list $L_{2,3}$ and $[l_2, r_2]$ to retrieve collective top-k answers as required. Since Rank-Join accesses the tuples in ranked order from the input lists, we do not have to physically merge tuples from the range $[l_1, l'_1 - 1]$ and $[r'_1 + 1, r_1]$. It only suffices to have the ability to retrieve the tuples in non-increasing order of the score from the logically merged list $L_{2,3}$. RMQ structure maintained on attribute α^s_1 of the relation R_1 can be used to achieve this goal along with a max-heap (binary search tree) which is initially empty as follows: We begin by initiating the ORR queries on ranges $[l_1, l'_1 - 1]$, $[r'_1 + 1, r_1]$ separately. The highest scored tuples obtained using ordered range retrieval from both the ranges are then inserted into the heap. Whenever Rank-Join tries to access a next tuple from list $L_{2,3}$ we do the following: (1) We perform the extract-max operation on the heap and supply it as a next tuple to Rank-Join. (2) We identify the range to which tuple just extracted belongs to and use the ORR query initiated on it to obtain next highest scored tuple, which is then inserted into the heap.

The process of merging subtasks 2 and 3 described above can also be applied to merge subtasks 4 and 5 in a similar way. Let the two new merged tasks be $merge_{2,3}$ and $merge_{4,5}$ respectively. Thus, Step 3 of the query algorithm executes two instances of Rank-Join algorithm applied to tasks $merge_{2,3}$ and $merge_{4,5}$.

Overlapping of subtasks: Though tasks $merge_{2,3}$ and $merge_{4,5}$ are independent of each other, we still need collective top-k answers and not top-k answers for these tasks individually. This allows us to overlap the execution of two tasks by executing them simultaneously. A simple way to achieve this is to perform one step of Rank-Join algorithm on two tasks alternately. At any point during the execution Rank-Join maintains a threshold which gives an upper-bound on the score of all join combinations not yet seen. Let $T_{2,3}$ be the current threshold for the task $merge_{2,3}$ and

$T_{4,5}$ be the same for task $merge_{4,5}$. Then score of any unseen join combination from either tasks is $T = max(T_{2,3}, T_{4,5})$. Hence, by terminating the execution of both instances of Rank-Join algorithm when we have k distinct join results with combined score higher than threshold T, correctness of the algorithm is ensured. However, instead of switching between the tasks $merge_{2,3}$ and $merge_{4,5}$ at every step, a score guided strategy is likely to give us top-k answers faster. If $T_{2,3} > T_{4,5}$ then more steps need to be performed for $merge_{2,3}$ to reduce the value of $T_{2,3}$ and, hence, the value of the threshold, leading to possible faster reporting of ranked join results.

Thus, in our final query algorithm, for Step 3 we execute two Rank-Join instances simultaneously on tasks $merge_{2,3}$, $merge_{4,5}$. We can optimize the query algorithm further as below. So far we have allowed Rank-Join to make only sorted accesses to the input. However our index also offers random access capabilities using its RORR structure component i.e., given two relations R_1 and R_2, our index can retrieve all the tuples from relation R_2 which produce valid join results (as well as satisfy given $select\text{-}predicate$, if any) for a given tuple $t_x \in R_1$ and vice a versa. As noted in [73], we can try to achieve better performance by allowing Rank-Join to exploit random access capabilities of our index. Precomputed answers obtained by querying score-matrix for subtask 1 can help us further to achieve early termination of Rank-Join instances i.e., we can terminate execution of both Rank-Join instances when we have found k distinct join results with score higher than the current threshold T coming from either task $merge_{2,3}$ or $merge_{4,5}$ or subtask 1.

Unified query algorithm: Query algorithms described in this section and in previous section can be used when two input lists to be joined are known to be positively correlated and anti-correlated respectively. However, absence of prior knowledge about correlation can lead to incorrect algorithm selection resulting in poor query performance. Below we describe a simple hybrid approach that combines the advantages of both query algorithms and can achieve competitive performance for all inputs. The new query algorithm behaves exactly like the algorithm just described for positively correlated data except one modification: In step 3 of the algorithm, if two Rank-Join instances on tasks $merge_{2,3}$, $merge_{4,5}$ do not terminate before scan depth of $c\sqrt{km}$ is reached for either of them, we terminate both the instance and being execution of Subtasks 2 to 5 as described in previous

section. Here, c is a user controlled parameter and m is the length of the shortest list out of four lists passed as input to tasks $merge_{2,3}$ and $merge_{4,5}$ i.e., $m = min(l'_i - l_i + r_i - r'_i, r'_1 - l'_1)$ for $i = \{1, 2\}$. The choice of the parameter $c\sqrt{km}$ is motivated by the fact that scan depth of $O(\sqrt{km})$ is required to find the top-k answers for the input lists of length m with uniformly random scores [74]. This suggests that scan depth of $c\sqrt{km}$ or more is a good indicator of data not being positively correlated and, hence, the switch to Subtasks 2 to 5 which are aimed to provide the performance guarantee in the worst case. Parameter c allows user to balance the performance degradation of hybrid solution with respect to the query algorithms tuned for positively and negatively correlated data primarily based on the efficiency of ORR structure.

3.5 Top-k Join Queries with More Than Two Relations

A binary pipeline is a common approach for answering top-k join queries involving many relations. For simplicity, we elaborate on how Rank-Join works for a top-k join on three input lists, say R_1, R_2 and R_3. Two Rank-Join instances progressively join the three inputs to produce valid join combinations. Bottom Rank-Join instance generates partial joins results by joining R_1 with R_2. At each step, top Rank-Join instance reads the next highest ranked tuple from R_3 and next highest ranked partial join result from bottom Rank-Join instance. Here one step of top Rank-Join instance can force bottom Rank-Join instance to perform as many steps as necessary to obtain next highest ranked partial join result. Thus, early termination of top Rank-Join instance can avoid significant computational efforts. Below we show how we can achieve such a early termination for the top-most Rank-Join instance using score-matrices component of our index. Though, we assume join involves three relations only, the proposed technique can be easily extended for generic queries.

Index structure: To support top-k join queries involving multiple relations we maintain binary tress, RORR and RMQ structures as before, whereas score-matrices component requires the following changes. Instead of a 2-dimensional matrix for a pair of heavy nodes in binary trees, we now maintain an 3-dimensional matrix, with each entry storing top-k answers among a triplet of heavy nodes, each from a binary tree corresponding to a different relation. As we have total c^6 different

combinations of selection and join attributes over 3 relations, we maintain as many score-matrices corresponding to these combinations for a fixed k occupying $O(c^6(n/g)^3k)$ space. Further, since we maintain score-matrices for each $k = 1, 2, 4, ...$, total space needed for all score-matrices, and hence, the total index space can be bounded as $O(cN)$ by choosing appropriate grouping parameter $g = (c^2n)^{1-1/3}(k \log n/3)^{1/3}$.

Query algorithm: We begin by querying the binary trees to obtain the range $[l_i, r_i]$ of tuples for each of the three relations satisfying the given $select\text{-}predicates$ and range splitting as described in Lemma 3.2. Let range $[l_i, r_i]$ be divided into $[l_i, l'_i - 1]$, $[l'_i, r'_i]$ and $[r'_i + 1, r_i]$. Since the first and the last subranges are small ($\leq g$), as before we call the tuples in those ranges as fringe tuples. Now the top-k answers such that none of the tuples in the answer triplet (t_x, t_y, t_z) is a fringe tuple can be directly obtained by querying appropriate score-matrix. Then, we only need to look for answer triplets where at least one of the tuple is a fringe tuple using the binary pipeline approach.

Below we modify the working of the top-most Rank-Join instance in the pipeline to compute only those result triplets which contain one or more fringe tuples. For three relations, this Rank-Join instance reads the tuples from R_3 and partial join results produced by Rank-Join instance below it. We split this top-most instance into two sub-instances, the one which operates on the inputs lists $R^f_{1,2}$, R_3 and the one that operates on the input lists $R^{nf}_{1,2}$, R^f_3. Here $R^f_{1,2}$ represents the partial join results of R_1 and R_2 such that at least one of the tuple in result pair is a fringe tuple and $R^{nf}_{1,2}$ represents the remaining partial results. Similarly R^f_3 represents the fringe tuples in R_3 qualifying the applied $select\text{-}predicate$. Lists $R^f_{1,2}$ and $R^{nf}_{1,2}$ are not computed entirely. Rather the output of Rank-Join instance just below the top-most are simply bifurcated based on the tuples involved in the partial join. Thus, at each step of top-most Rank-Join instance, we retrieve next highest ranked partial join result for R_1 and R_2, then based on the list it belongs to we execute one of its two sub-instance and compute threshold for the upper bound on the score of any unseen join combination. We terminate the top-most Rank-Join instance when we have found k join results with the score higher than the threshold. Since we already have top-k answers computed from score-matrices, top-most Rank-Join instance can be terminated earlier than it would have been otherwise.

3.6 Experimental Analysis

In this section, we compare our proposed index with Rank-Join algorithm through experimentation. We first describe our experimental setup. Then, we compare the performance of our index with Rank-Join by varying different experimental parameters. We choose Rank-Join for comparative study because it was shown to have good performance in practice, and its variants HRJN, HRJN* are **instance optimal** with an optimality ratio of 2 for our experimental settings [44]: (1) join involves only two relations (2) aggregation function F depends on only one attribute from each relation. This optimality assumes that each relation R_i is accessed in non-increasing order of the score. Such an access model has been a common assumption in earlier studies of rank joins as well.

3.6.1 Experimental Setup

We consider two variants of the Rank-Join algorithm to compare our index against. The first variant which we call NRA is essentially HRJN* algorithm in [73] which is restricted to sorted accesses only. The other variant RA is Rank-Join algorithm capable of performing random accesses. We consider two variants of our index as well, IND-N and IND-P. IND-N is the index designed to provide performance guarantees as summarized in Theorem 3.4. For handling positively correlated data as described in Section 3.4, IND-P makes use of a variant of Rank-Join algorithm that utilizes random access (i.e., algorithm RA). Though any heuristic can be employed as pointed out earlier, we select RA as it can exploit the random access capabilities supported by our index and is known to outperform NRA for low join selectivity. We implemented all of the query algorithms using the programming language C++, compiled with the g++ compiler version 4.2. Our experiments were run on an Intel Core 2 Duo 2GHz machine (MAC OS 10.7.4) with a 8GB RAM.

For the experiments, we used both synthetically generated and real data. The first real data set NBA (`www.databasebasketball.com`) contains ≈20,000 statistics of an NBA player's performance. Our second data set, XML [127], consists of 160,000 tuples. It is the outcome of the join of two data sets recording the size and outdegree of a collection of XML documents. We use synthetic datasets of two relations with each relation having three attributes: select, join, score.

While select attributes are random number, join attributes are determined based on desired selectivity. By default we assume that the tuples in two relations have one-one mapping. In this case we assign distinct join values in the range $[1, n]$ for each tuple in both relations. For generating datasets with varying correlation we use a correlation parameter ρ (-1 $\leq \rho \leq$ 1) and generate the score values of tuples as follows. We generate a pair of correlated random numbers for each join value, using equations $X_2 = X_1$, $Y_2 = \rho X_1 + \sqrt{1-\rho^2} Y_1$ where X_1, Y_1 are input random numbers in the range $[1, n]$ and X_2, Y_2 follows given correlation parameter ρ. Values of X_2 and Y_2 can be thought of as temporary scores assigned to tuples, and thus, decide their ordering within the individual relations. Since X_2 and Y_2 are correlated, positions of tuples sharing a join value in two relations also follows same correlation, for instance, for $\rho = 1$ all join values will be located at same position in both the relations that are sorted based on score. By varying the value of ρ, we can create lists with positive or negative and stronger or weaker correlations. After setting the positions of tuples in two relations, we generate the score values for tuples in each relation as per the desired probability distribution [3]. We use uniform, gaussian (mean = 0, standard deviation = n/4), zipf ($\theta = 0.7$) for our experiments. Since our index achieves similar performance for these distributions, we report results for uniform score distribution only.

To provide higher selectivity, we begin by generating two relations with one-one tuple mapping as before. We can not randomly replicate the join values in the relations as it would hurt the correlation. Inspired from [3], we use the correlation parameter ρ to control the replication of the join values. We pre-define a set of distinct join values whose cardinality depends on given selectivity. These values are placed in the first relation at equidistant positions. Let m be one such position. We randomly replicate the join value at position m for the tuples in the range $[m - n\rho, m + n\rho]$ as many times as necessary. Whenever a tuple t_x in first relation receives the updated join value from pre-defined set, tuple t_y in second relation that shares the old join value of t_x also updates its join value. There will be exactly one such tuple as the original relations have one-one mapping.

In our tests, we use following default settings for different experimental parameters: (a) n = 100,000 (b) k = 10 (c) Join selectivity = 0.001% (one-one mapping) (d) $\rho = -0.8$, 0.6 (d)

Scoring function = SUM. For each dataset we generate a set of 100 queries at random ensuring that more than $2g$ tuples qualify the selection criteria in both relations based on number of required answers. Average time required to answer such a set of queries is used as a measure to evaluate the performance of different query algorithms. We justify this selection criteria later in the Section 3.6.3. In addition, we also ensure that the queries can not be answered only using score-matrices so as to avoid any unfair advantage to the proposed index against different variants of Rank-Join algorithm.

3.6.2 Effects of Correlation

In this experiment, we compare the performance of our index with Rank-Join by varying the correlation between the two relations. Rank-Join assumes the availability of access to tuples in a ranked order. Hence, we further differentiate each of the Rank-Join variant under consideration based on how such sorted access is provided. Algorithms SORT-NRA and SORT-RA sort the tuples qualifying the select predicates. For fair comparison, we also consider algorithms ORR-NRA and ORR-RA which utilize the ORR structure described in Section 2.1. As random accesses are know to help Rank-Join algorithm to terminate faster for lower selectivity, we omit results of SORT-NRA and ORR-NRA for better clarity in this experiment. We also consider a variant IND-H that is essentially the same as IND-P, however, it employes a unified query algorithm as explained earlier.

All the Rank-Join algorithms show performance improvement as we vary ρ from -1 to 1 (Figure 3.2) since top-k answers can be found at smaller scan depths. For negatively correlated data, Rank-Join needs to scan through a significant portion of the input lists. Retrieving all these tuples by ORR query poses overhead which can be longer than time required for sorting the qualifying tuples. Whereas for positively correlated data sorting proves to be a performance bottleneck. Hence, for the remaining experiments we use SORT version of the Rank-Join algorithms for negatively correlated data and ORR version otherwise.

As shown in Figure 3.2 performance of IND-N remains unaffected due to variations in ρ. This helps IND-N to outperform all Rank-Join variants when input lists are negatively correlated. Even for the variant of Rank-Join that performs best IND-N offers performance improvement up to the factor of 8. Even higher performance gain will be obtained when more number of tuples qualify

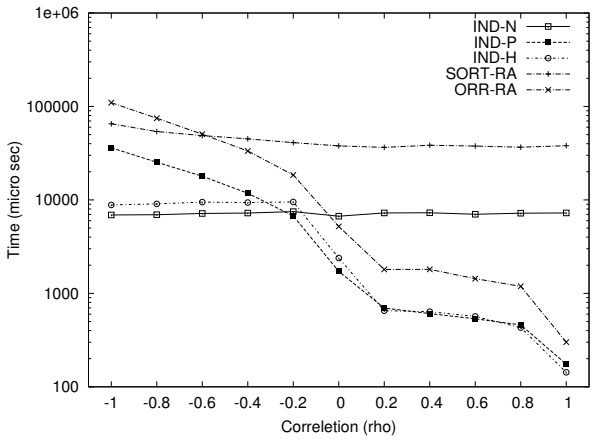

FIGURE 3.2. Effect of correlation (ρ)

query predicates (Section 3.6.4). Algorithms employing tighter terminating conditions than used in Rank-Join also can be forced to scan through the number of tuples proportional to n when data is negatively correlated, whereas IND-N can return top-k by performing $\tilde{O}(\sqrt{kn})$ join trials only. Thus, IND-N offers a solution for handling worst case scenarios where heuristic approaches are known to be inefficient. However, insensitivity of IND-N makes it a less attractive option under more favorable conditions. Algorithm of IND-P can help us to maintain the competitive edge over the Rank-Join algorithms even for the positively correlated data. Since IND-P makes use of a heuristic algorithm at the core, its performance deteriorates with decreasing ρ. IND-H i.e., a hybrid query algorithm which adapts to the underlying correlation and remains competitive for the entire range of ρ, provides us with a unified way of querying the data. Figure 3.2 shows the performance of this unified query algorithm with $c = 1$.

3.6.3 Effects of k

We now study the effect of the number of required answers on performance. Figure 3.3 shows how the query time increases with increasing k up to 100 for two datasets with ρ = -0.8 and 0.6. The query time of all Rank-Join variants increases with k because more tuples are required to be checked in order to obtain the top-k join results. However, the increase is very small for negatively correlated dataset. When two lists are opposingly ranked, even for small value of k, Rank-Join has to scan significant portion of the lists. As a result, when Rank-Join terminates for a top-k

query, it is highly likely that it has also seen the $(k+1)$th result. The query time of IND-N and IND-P shows an interesting step property for increase in k. Since we store the partial answers in score-matrices for $k = 1, 2, 4, 8, ...$ and we probe score-matrix which stores top-(2^m) answers such that $2^{m-1} < k \leq 2^m$, grouping factor which directly controls the query time remains same for all values k between 2^{m-1} and 2^m. For negatively correlated data, since IND-N spends fixed amount of efforts based on the grouping factor, we get flat performance between two values of k which are consecutive powers of 2. However, for positively correlated data, Rank-Join heuristic employed by IND-P spends efforts proportional to scan depth which increases with increase in k. This leads to increase in query time of IND-P even for values of k between consecutive powers of 2. Whenever k is equal to power of 2, query time for both IND-N, IND-P increases substantially due to sudden increase in the grouping factor leading to step graph-like behavior.

For negatively correlated data IND-N offers significant performance gains over the Rank-Join variants for small values of k. With increase in k, query time for IND-N increases more rapidly than that of Rank-Join, thus, gradually diminishing the advantages it offers. We notice that the performance gains offered by IND-N and IND-P are due to the partially stored answers in score-matrices. As k increases, at some point the parameter $2g$ becomes larger than the number of tuples qualifying the query predicates. When this happens we may not be able to find any stored answers that are useful for answering the query i.e., even IND-N will have to access all the tuples in the input lists. However the exact breaking point at which IND-N offers no performance gain depends on the

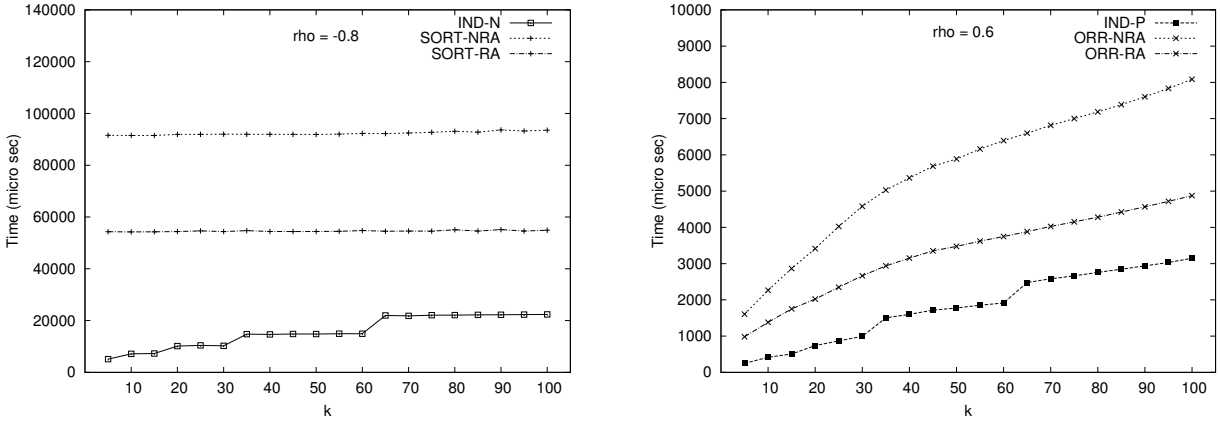

FIGURE 3.3. Effect of k

factors like efficiency of ORR query, heap implementation and cost of random access etc. To avoid such a scenario we ensure that query predicates return more than $2g$ tuples, as pointed out earlier. In practice, Rank-Join algorithms provide reasonable performance for these cases because under such circumstances either k is proportional to n or input lists are small (less than $\tilde{O}(\sqrt{n})$). For positively correlated data, IND-P remains competitive even for higher values of k, as it takes advantage of stored answers at the same time also gets benefited by friendly correlation between input lists.

3.6.4 Effects of Number of Tuples

We now vary the number of tuples qualifying the select predicates, and investigate its effect on performance. Figure 3.4 shows how query time increases for both Rank-Join variants with increasing number of tuples up to 90,000. Increasing the number of data items has a considerable impact on the performance of Rank-Join algorithm whereas IND-N remains almost insensitive. For negatively correlated data, the scan depth increases linearly along with number of tuples in the input lists adversely affecting the performance of Rank-Join algorithms. On the other hand, IND-N only needs to look at more number of entries in the score-matrix to cope up with the increase in input size. For retrieving top-10 tuples from the input lists of size 90,000 that are negatively correlated, IND-N outperforms the variant of Rank-Join that performs best by a factor of 10. We do not show the results of this experiment for positively correlated data as query time increases only marginally with more tuples in input lists. We note that performance of IND-N is more dependent on the database size than the number of tuples in the inputs lists to be joined.

3.6.5 Effects of Join Selectivity

In this experiment, we fix the value of k at 10 and vary the join selectivity gradually up to 1%. With increase in selectivity, performance of RA algorithm degrades as shown in Figure 3.5. Degradation of RA is severe for negatively correlated data as number of valid join combinations evaluated by RA increases linearly with selectivity. For positively correlated data, though RA shows improvements initially, eventually its query time begins to increase. Both the variants of our index shows increasing query times for higher selectivity. During query RORR structure step of

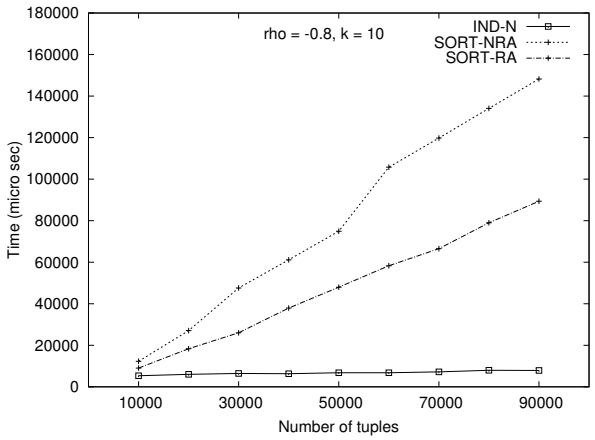

FIGURE 3.4. Effect of number of tuples

the algorithm, IND- N performs a binary search on valid join combinations of a particular tuple (Lemma 2.2) to filter out the ones which do not qualify the query predicates. Increase in query time for IND-N can be attributed to higher cost of binary searches with increase in selectivity. For IND-P such increase results due to deteriorating performance of heuristic RA that is being used internally.

NRA stands out from the other algorithms as it gets more and more efficient with increasing selectivity. Performance degradation issue of IND-P can be eliminated by using NRA instead of RA as a part of query algorithm. Increase in selectivity essentially lessens the impact correlation has on the top-k join query processing. Since for higher selectivity, negative correlation is no longer a concern, IND-N can be replaced by its counterpart IND-P which can benefit from efficiency of NRA algorithm.

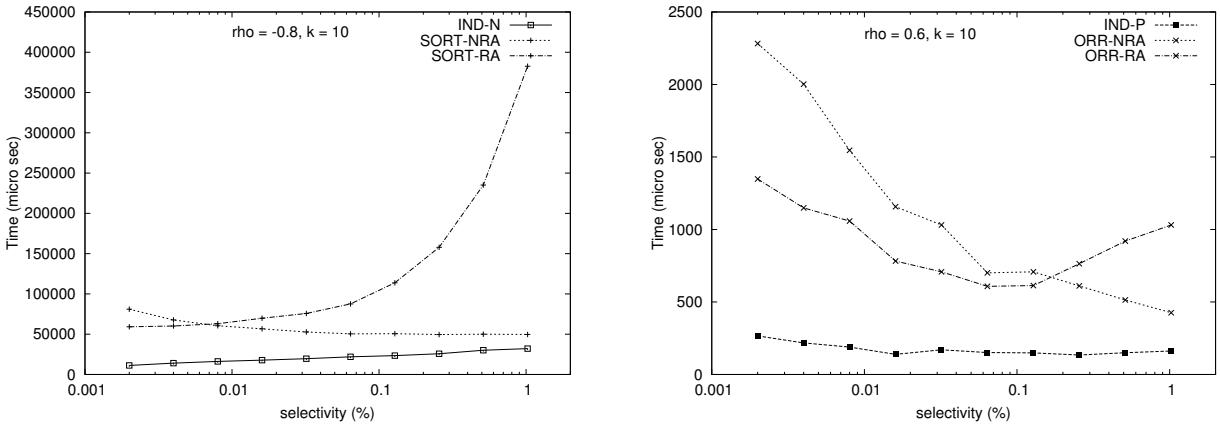

FIGURE 3.5. Effect of join selectivity

3.6.6 Results for Real Datasets

We now evaluate performance of IND-N for two real datasets. We use *field goals made/field goals attempted* and *outdegree/size* as a ranking function for NBA and XML data respectively. Moreover, we do not apply any select predicates i.e., we retrieve top-k tuples over the entire datasets. Instead of comparing the query times, total number of sorted and random accesses done by the algorithm is used as a measure of performance. This helps us to analyze the comparative performance independent of the system and implementation details. Figure 3.6 shows results of the experiment which are similar to the results obtained for synthetic dataset with negative correlation in Figure 3.3. For both the datasets, IND-N performs fewer accesses than both NRA and RA. As observed earlier, for lower values of k performance gap between IND-N and Rank-Join is substantial, and it narrows down gradually with increase in k. Experiment with real datasets reveal an interesting fact that though we expect query time and total number of accesses made by IND-N to increase with k, occasionally query performance may improve for higher value of k. Let S_1 be the set tuples part of a group for $k_1 = 2^{m-1}$ (Section 3.3.1) and similarly S_2 be the set for $k_2 = 2^m$. Then under following circumstances such a behavior can be observed: (1) Set S_1 is a proper subset of S_2, (2) All the tuples in set S_2 qualify the applied select predicate, and (3) Number of tuples qualifying the select predicate are strictly less than $2|S_1|$.

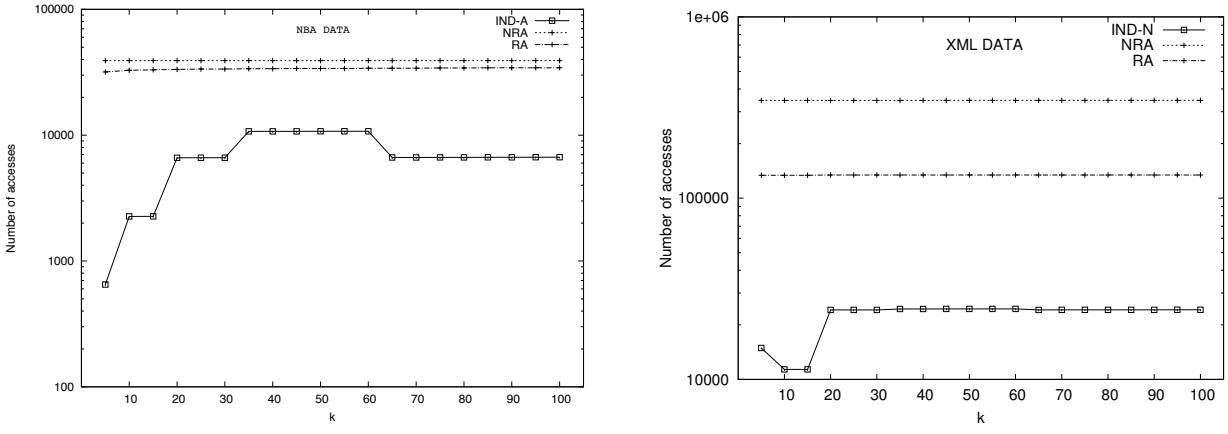

FIGURE 3.6. Results for real datasets

3.6.7 Index Construction Time

In the final set of experiments, we study the effect of data size, join selectivity and correlation on time required to build the proposed indexing structure. Dependance of index construction time on data size ($\tilde{O}(n^{3/2})$) is evident in the Figure 3.7, which shows the time required to build an index for n varying from 100,000 to 1,000,000 with default values of correlation (ρ =-0.8) and selectivity. We choose not to build score-matrix for k such that grouping parameter $g \geq n$, as under such circumstances the proposed data structures do not offer any benefits over algorithmic approach. We observe that with increasing value of k, group size g increases and time required to populate a score-matrix i.e., $O(n^2 \log n/g)$ decreases. With score-matrix computations dominating the index construction time, we may also choose not to build score-matrices for smaller values of k by sacrificing the query time to some extent. By excluding score-matrices only for $k = 1, 2$ we can achieve up to 45% reduction in construction time. Such index only doubles the query time in the worst case by returning top-4 answers even when top-1 or top-2 results are requested. "Querying ORR Structure" being a common link between index construction and query execution of IND-N, index construction time shows trends similar to the query performance of IND-N for varying correlation and selectivity. This also suggests that the ideas introduced in Section 3.4 to take advantage of favorable inputs can be used to improve the construction time as shown in Figure 3.7 for $\rho = 0.6$.

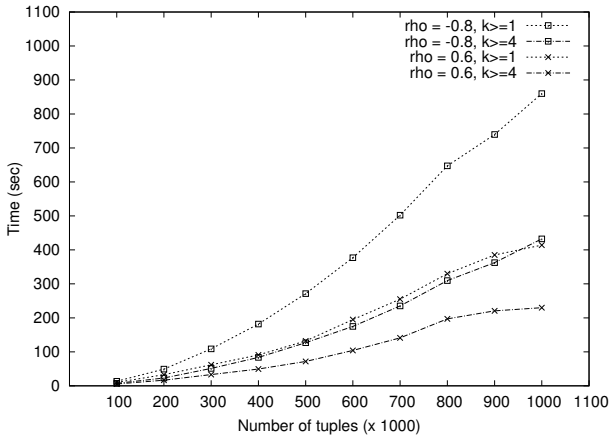

FIGURE 3.7. Index construction time

3.7 Related Work

Top-k queries on a traditional dataset have been well studied in the literature. Fagin [39] first investigated the problem of answering top-k selection queries over pairwise combinations of multiple ranked inputs and proposed an algorithm for this problem assuming both sorted and random accesses are available for all the inputs. Later Fagin et al. [41], Guntzer et al. [57], Nepal and Ramakrishna [105] independently proposed threshold algorithm TA improving upon the earlier Fagin's algorithm. Several extensions of TA have been proposed for processing top-k queries in different environments [22, 96, 13, 36, 137, 4]. As the size of the inputs grows, random access to each of them becomes a bottleneck for query performance. To remedy this, trade-off between sorted access and random access has been studied by Fagin et al. [41], Chang and Hwang [25], and Bruno et al. [20]. Algorithms have also been proposed to account for the the case when random accesses are not supported or are extremely expensive [41, 72, 58].

Natsev et al. [100] proposed algorithm J* for efficient processing of top-k join queries over ranked inputs with any arbitrary join conditions. In [25] Chang and Hwang extend their algorithm for top-k selection queries to answer top-k join queries. Extending the work on top-k selection queries, Ilyas et al. [73] gave Rank-Join algorithm for top-k join queries. Even though experimental studies in [73] show that Rank-Join significantly outperform J*, as pointed in [93, 3, 44] Rank-Join can access more objects than necessary for answering the query because of it's lazy stopping condition. Recently proposed algorithms LARA-J [93], NR-JTop [3], FRPA [44] employ efficient stopping mechanism and are shown to outperform Rank-Join. However, despite tighter threshold used in stopping mechanism, these algorithms may also have scan depth proportional to input size in the worst case scenarios. Efforts have been made towards extending the relational algebra in [88] so as to support efficient evaluation of top-k join queries. Li et al. [87] combines top-k join processing with aggregate queries. Index-based approaches for answering the top-k queries have been presented in [127, 126]. The index proposed in [126] is applicable only for top-k selection queries, whereas the index proposed in [127] makes use of a predefined number K and cannot answer top-k join queries with $k > K$.

3.8 Summary

It is known that for uniformly random scores between two relations of length n, scan depth of $O(\sqrt{kn})$ is required while answering the top-k join query. However, in the worst-case scenario if two relations are inversely ranked then one might need scan depth proportional to the size of input relations. In many situations, when users want to optimize between multiple criteria of selections, these criteria are often inversely correlated. In this chapter, we proposed the indexing technique for achieving sub-linear worst case query time for answering top-k join queries involving two relations while keeping space requirement linear to the size of the database. Thus, we get the average case performance even in the worst-case scenario.

Chapter 4
Inverted Indexes for Phrases and Strings

4.1 Introduction

The most popular data structure in the field of Information Retrieval is the inverted index. For a given collection of documents, the index is defined as follows: Each word in this collection is called a *term* and corresponding to each term we maintain a list, called *inverted list*, of all the documents in which this word appears. Along with each document in this list we may store some score which indicates how important the document is with respect to that word. Different variants of the inverted index sort the documents in the inverted lists in a different manner. For instance, the sorting order may be based on the document ids or the scores. Compression techniques are often applied to further reduce space requirement of these lists. However, inverted index has a drawback that it can support queries only on predefined words or terms. As a result, it cannot be used to index documents without well-defined word boundaries.

Different approaches have been proposed to support phrase searching using an inverted index. One strategy is to maintain the position information in the inverted list, that is, for each document d in the inverted list of a word w, we store the positions at which w occurs in d. The positions corresponding to each d in the list can be sorted so as to achieve compression (using encoding functions like gap, gamma, or delta) [59]. To search a phrase, we first search for all the words in the phrase and obtain the corresponding inverted lists. The positions of each word within a document are extracted, so that we can then apply an intersection algorithm to retrieve those documents where these words are appearing in the same order as in the phrase. Another (naive) approach is to store inverted lists for all possible phrases, however, the resulting index size will be very large thus prohibiting its use in practice [140]. Different heuristics are proposed in this respect, such as maintaining the inverted lists only for popular phrases, or maintaining inverted lists of all phrases up to some fixed number (say h) of words. Another approach is called "next-word

index" [134, 10, 11, 135], in which corresponding to each term w, a list of all the terms which occurs immediately after w is maintained. This approach will double the space, but it can support searching of any phrase with two words efficiently. Nevertheless, when the phrase goes beyond two words, we have to fall back to the intersection algorithm.

In this chapter, we first introduce a variant of inverted index which naturally works for string as well as phrase searching. Our index does not assume any restrictions on the length or the popularity of the phrases. In addition, by avoiding the use of the intersection algorithm we achieve provable bounds for the query answering time with respect to the output size. Furthermore, we show different heuristics and compression techniques to make our index space-efficient.

4.2 Theoretical Framework

In traditional inverted indexes, phrase queries are performed by first retrieving the inverted list for each word in the phrase and then applying an intersection algorithm to retrieve those documents in which the words appear in the same order as in the phrase. Unfortunately, there is no efficient algorithm known which performs this intersection in time linear to the size of the output. Another limitation of the traditional inverted indexes is that they do not support string documents where there is no word demarcation (a query pattern can begin and end anywhere in the document). A naive approach to address these issues is to maintain inverted lists for all possible phrases (or strings). In the next subsection, we introduce a simple index that is based on a suffix tree and augments this with the inverted lists. This index can answer the queries in optimal time, however, the space is a factor of $|\mathcal{D}|$ away from the optimal. As phrase is a special case of a string (that is, string that starts and ends at word boundaries), we will explain our indexes in terms of strings.

4.2.1 Inverted Lists

Let $\mathcal{D}$=$\{d_1, d_2, ..., d_{|D|}\}$ be the collection of documents of total length n drawn from an alphabet set Σ, and Δ be the generalized suffix tree of $\mathcal{D}$. Let u be the locus node of a pattern P. Now a naive solution is to simply maintain an inverted list for the pattern corresponding to $path(u)$ for all internal nodes u in Δ. The list associated with a node u consists of pairs of the form $(d_j, score(path(u), d_j))$

for $j = 1, 2, 3, ..., |\mathcal{D}|$, where the score of a document d_j with respect to pattern $P = path(u)$ is given by $score(path(u), d_j)$. We assume that such a score is dependent only on the occurrences of P in the document d_j. An example of such a score metric is frequency, so that $score(P, d_j)$ represents the number of occurrences of pattern P in document d_j. For a given online pattern P, the top-k highest scoring documents can be answered by reporting the first k documents in the inverted list associated with the locus node of P, when the inverted lists are sorted by score order. Since the inverted list maintained at each node can be of length $|\mathcal{D}|$, the total size of this index is $O(n|\mathcal{D}|)$. Though this index offers optimal query time, it stores the inverted list for all possible strings. In the next subsection we show how the inverted lists can be stored efficiently in a total of $O(n)$ space.

4.2.2 Conditional Inverted Lists

The key idea which leads to $O(n)$ storage for inverted lists is the selection of nodes in the suffix tree for which inverted lists are actually maintained. We begin with the following definitions:

- Maximal String: A given string P is *maximal* for document d, if there is no other string Q such that P is a prefix of Q and every occurrence of P in d is subsumed by Q.
- Conditional Maximal String: Let Q be a maximal string for which P is a prefix and there is no maximal string R such that R is in between P and Q, that is, P is a prefix of R and R is a prefix of Q. Then we call Q a *conditional maximal string* of P.

Consider the following sample documents d_1, d_2, and d_3:

- d_1: *This is a cat. This is not a monkey. This is not a donkey.*
- d_2: *This is a girl. This is a child. This is not a boy. This is a gift.*
- d_3: *This is a dog. This is a pet.*

Note that "*This is* " is maximal in d_1 as well as d_2, but not in d_3. The conditional maximal strings of "*This is* " in d_1 are "*This is a cat ... donkey.*" and "*This is not a* ". The conditional maximal strings of "*This is* " in d_2 are "*This is a* " and "*This is not ... gift.*".

Lemma 4.1. The number of maximal strings in a document d_j is less than $2|d_j|$.

Proof. Consider the suffix tree Δ_j of document d_j. Then for each maximal string P in d_j, there exists a unique node u in Δ_j such that $path(u) = P$. Thus, the number of maximal strings in d_j is equal to the number of nodes in Δ_j. □

Lemma 4.2. For a given pattern P, we have $score(P, d_j)$ = $score(P_i, d_j)$, where P_i is the shortest maximal string in d_j with P as prefix. If such a string P_i does not exist, then $score(P, d_j) = 0$.

Proof. As P_i is the shortest maximal string in d_j with P as prefix, every occurrence of a pattern P in d_j is subsumed by an occurrence of P_i. Hence, both patterns will have same score with respect to document d_j, with $score(P, d_j) = 0$ signifying that the pattern P does not occur in d_j. □

Lemma 4.3. For every maximal string $Q(\neq empty\ string)$ in d_j, there exists a unique maximal string P such that Q is a conditional maximal string of P.

Proof. Corresponding to each maximal string Q in d_j, there exists a node u in Δ_j (suffix tree of document d_j) such that $Q = path(u)$. The lemma follows by setting $P = path(parent(u))$, where $parent(u)$ denotes the parent of u in Δ_j. □

The number of maximal strings in $\mathcal{D}$=$\{d_1, d_2, ..., d_{|D|}\}$ is equal to the number of nodes in Δ (Lemma 4.1). In the context of maximal strings, the index in Section 4.2.1 maintains inverted lists for all maximal strings in $\mathcal{D}$. However, $score(P, d_j)$ depends only on pattern P and document d_j. This gives the intuition that, for a particular document d_j, instead of having entries in inverted lists corresponding to all maximal strings in $\mathcal{D}$, it is sufficient to include d_j in the inverted lists of only those strings which are maximal in d_j. Thus, for each document d_j, there will be at the most $2|d_j|$ entries in all inverted lists, so that the total number of such entries corresponding to all documents is at most $\sum_{j=1}^{|\mathcal{D}|} 2|d_j| = O(n)$. However, the downside of this change is that the simple searching algorithm used in Section 4.2.1 can no longer serve the purpose. Therefore, we introduce a new data structure called "conditional inverted lists" which is the key contribution.

From now onwards, we refer to the maximal strings by the pre-order rank of the corresponding node in Δ. That is $P_i = path(u_i)$, where u_i is a node in Δ with pre-order rank i. In contrast to the

traditional inverted list, the conditional inverted list maintains $score(P_i, d_j)$ only if P_i is maximal in d_j. Moreover $score(P_i, d_j)$ is maintained not with P_i, but instead with P_x, such that P_i is a conditional maximal string of P_x in d_j. Therefore, u_x will be a node in the path from root to u_i. Formally, the conditional inverted list is an array of triplets of the form $(string\ id, document\ id, score)$ sorted in the order of string-ids, where the string-id is pre-order rank of a node in Δ. A key observation is the following: The conditional inverted list of a string P_x has an entry $(i, j, score(P_i, d_j))$ if and only if P_i is a conditional maximal string of P_x in document d_j. From the earlier example, the conditional inverted list of "*This is* " has entries corresponding to the following strings. We assign a *string id* to each of these strings (for simplicity) and let the $score$ of a string corresponding to a document be its number of occurrences in that document.

"*This is a cat ... donkey.*" (*string id* = i_1, $score$ in $d_1 = 1$)

"*This is not a* " (*string id* = i_2, $score$ in $d_1 = 2$)

"*This is a* " (*string id* = i_3, $score$ in $d_2 = 3$)

"*This is not a ... gift.*" (*string id* = i_4, $score$ in $d_2 = 1$)

Since the *string ids* are based on the lexicographical order, $i_3 < i_1 < i_2 < i_4$. Then the conditional inverted list associated with the string "*This is* " is given below. Note that there is no entry for d_3, since "*This is* " is not maximal in d_3.

string id	i_3	i_1	i_2	i_4
document id	d_2	d_1	d_1	d_2
score	3	1	2	1

We also maintain an RMQ (range maximum query) structure over the $score$ field in the conditional inverted lists so as to efficiently retrieve documents with the highest score. We begin by retrieving document with the highest score using the suffix range as input. Such a document partitions the suffix range into two subranges which are then used as input for RMQ to obtain the document with next highest score. We elaborate on such recursive applications of RMQ later in following subsection.

Lemma 4.4. The total size of conditional inverted lists is $O(n)$.

Proof. Corresponding to each maximal string in d_j, there exists an entry in the conditional inverted list with document id j. Hence, the number of entries with document id as j is at the most $2|d_j|$ and the total size of conditional inverted lists is $O(\sum_{j=1}^{|\mathcal{D}|} 2|d_j|) = O(n)$. □

Lemma 4.5. For any given node u in Δ and any given document d_j associated with some leaf in the subtree of u, there will be exactly one string P_i such that (1) P_i is maximal in d_j, (2) $path(u)$ is a prefix of P_i, and (3) the triplet $(i, j, score(P_i, d_j))$ is stored in the conditional inverted list of a node $u_x \neq u$, where u_x is some ancestor of u.

Proof. Since there exists at least one occurrence of d_j in the subtree of u, Statements $(1), (2)$, and (3) can be easily verified from the definition of conditional inverted lists. The uniqueness of P_i can be proven by contradiction. Suppose that there are two strings $P_i^{'}$ and $P_i^{''}$ satisfying all of the above conditions then $path(u)$ will be a prefix of $P_i^* = lcp(P_i^{'}, P_i^{''})$, where lcp is the longest common prefix. Then from the one-to-one correspondence that exists between maximal strings and nodes in suffix tree (Lemma 4.1), it can be observed that the lcp between two maximal strings in a document d_j is also maximal. Thus P_i^* is maximal in d_j and this contradicts the fact that, when $P_i^{'}$ (or $P_i^{''}$) is a conditional maximal string of P_x, there cannot be a maximal string P_i^*, such that P_i^* is a prefix of $P_i^{'}$ and P_x is a prefix of P_i^*. □

4.2.3 Answering Top-k Queries

Let P be the given online pattern of length p. To answer a top-k query, we first match P in Δ in $O(p)$ time and find the locus node u_i. Let $\ell = i$ and r be the pre-order rank of the rightmost leaf in the subtree of u_i, that is, P_ℓ and P_r represent the lexicographically smallest and largest maximal strings in $\mathcal{D}$ with $path(u_i)$ as a prefix, then, all maximal strings with P as prefix can be represented by P_z, $\ell \leq z \leq r$. From Lemmas 4.3 and 4.5, for each document d_j which has an occurrence in the subtree of u_i, there exists a unique triplet with score $score(P, d_j)$ in the conditional inverted list of some ancestor node u_x of u_i with $string\ id \in [\ell, r]$. Now the top-k documents can be retrieved by first identifying such triplets and then retrieving the k highest scored documents.

Note that the triplets in the conditional inverted lists are sorted according to the string-ids. Hence, by performing a binary search of ℓ and r in the conditional inverted list associated with each ancestor of u_i, we obtain t non-overlapping intervals $[\ell_1, r_1], [\ell_2, r_2], ..., [\ell_t, r_t]$, where $t < p$ is the number of ancestors of u_i. Using an RMQ (range maximum query) structure over the $score$ field in the conditional inverted lists, the k triplets (thereby documents) corresponding to the k highest scoring documents can be retrieved in $O(t + k \log k)$ time (Lemma 2.1). Hence the total query time is $O(p) + O(t \log n) + O(t + k \log k) = O(p \log n + k \log k)$.

Theorem 4.6. The String Inverted Index for a collection of documents $\mathcal{D} = \{d_1, d_2, ..., d_{|D|}\}$ of total length n can be maintained in $O(n)$ space, such that, for a given pattern P of length p, the top-k document queries can be answered in $O(p \log n + k \log k)$ time.

Note that the same structure can be used for document listing problem [99], where we need to list all the documents which has an occurrence of P. This can be answered by retrieving all the documents corresponding to the intervals $[\ell_1, r_1] \cup [\ell_2, r_2] \cup ... \cup [\ell_t, r_t]$ in the conditional inverted lists. Hence the query time is $O(p \log n + docc)$, where $docc$ is the number of documents containing P. If our task is to just find the number of such documents (counting, not listing), we may use $docc = \sum_{i=1}^{t} (r_i - \ell_i)$, and can answer the query in $O(p \log n)$ time.

Theorem 4.7. Given a query pattern P of length p, the document listing queries for a collection of documents $\mathcal{D} = \{d_1, d_2, ..., d_{|D|}\}$ of total length n can be answered in $O(p \log n + docc)$ time, where $docc$ is the number of documents containing P. The computation of $docc$ (document counting) takes only $O(p \log n)$ time.

The index described in this section so far is a generalized index for string documents. When word boundaries are well-defined and query patterns will be aligned with word boundaries as well, we can build the inverted index for phrases by replacing the generalized suffix tree with a word suffix tree. A word suffix tree is a trie of all suffixes which start from a word boundary. We call this a phrase inverted index. Theorems 4.6 and 4.7 can be rewritten for phrase inverted index as follows:

Theorem 4.8. The Phrase Inverted Index for a collection of documents $\mathcal{D} = \{d_1, d_2, ..., d_{|D|}\}$ with total N suffixes, which start from a word boundary, can be maintained in $O(N)$ space, such that, for a given pattern P of length p, the top-k, document listing, and document counting queries can be answered in $O(p \log N + k \log k)$, $O(p \log N + docc)$ and $O(p \log N)$ time, respectively.

4.3 Practical Frameworks

In Section 4.2, we introduced the theoretical framework for our index. However, when dealing with the practical performance, the space and time analysis has to be more precise than merely a big-O notation. Consider a collection of English text documents of total length n, where each character can be represented in 8 bits then the text can be maintained in $8n$ bits. The conditional inverted list can consist of at the most $2n$ triplets and if each entry in the triplet is 32 bits (word in computer memory), then the total size of the conditional inverted lists can be as big as $(2n \times 3 \times 32)$ bits $= 24 \times (datasize)$. Moreover, we also need to maintain the generalized suffix tree, which takes $\approx$20-30 times of the text size. Hence the total index size will be $\approx 50 \times (datasize)$. This indicates that the hidden constants in big-O notation can restrict the use of an index in practice.

In this section, we introduce a practical framework of our index when frequency is used as score metric, that is, $score(P, d_j)$ represents the number of occurrences of pattern P in document d_j. However, the ideas used can also be applied for other measures. Based on different tools and techniques from succinct data structures, we design three practical versions of our index (index-A, index-B, index-C) each successively improving the space requirements. We try to achieve the index compression by not sacrificing too much on the query times. Index-C takes only $\approx 5 \times (datasize)$, and even though it does not guarantee any theoretical bounds on query time, it outperforms the existing indexes [33] for top-k retrieval.

4.3.1 Index-A

Index-A is a direct implementation of our theoretical index from Section 4.2 with one change. As suffix tree is being used as an independent component in the proposed index, we replace it by compressed suffix tree (CST) without affecting the index operations and avoid the huge space

required for suffix tree. We treat index-A as our base index as it does not modify the conditional inverted lists which form the core of the index.

4.3.2 Index-B

In this version, we apply different empirical techniques to compress each component of the triplets from the conditional inverted list separately.

Compressing document array: Taking into account the fact that the total number of documents is $|D|$, we use only $\lceil \log |D| \rceil$ bits (instead of an entire word) per entry for the document value.

Compressing score array: When pattern frequency is used as the score metric, score array consists of numbers ranging from 1 to n. The most space-efficient way to store this array would be to use exactly the minimal number of bits for each number with some extra information to mark the boundaries. But this approach may not be friendly in terms of retrieving the values. Our statistical studies showed that more than 90% of entries have frequency values less than 16 (which needs only 4 bits). This leads us to the heuristic for distributing frequency values into four categories: *a*) 1-4 bits, *b*) 5-8 bits, *c*) 9-16 bits, and *d*) 17-32 bits based on the actual number of bits required to represent each value. We use a simple wavelet tree structure [55] which first splits the array into two arrays, one with 1-8 bits and another with 9-32 bits, required per entry. Both arrays are further divided to cover the categories a, b and c, d, respectively. Each of the child nodes can be further divided into two. The values stored at the leaf nodes of the wavelet tree take only as many bits as represented by the category it belongs to. Further, we use rank-select [97, 116] structures on the bit vectors in the wavelet tree for fast retrieval of values.

Compressing string-id array: Since the entries in the conditional inverted lists are sorted based on string-id values, we observe that there will be many consecutive entries of the same string-id, each with different document-id. Therefore, run-length encoding is a promising technique for string-id compression. In order to support fast retrieval of a particular string-id value, we again maintain additional bit vectors to keep track of which string-id values are stored explicitly and which values are eliminated due to repetition in the conditional inverted lists.

4.3.3 Index-C

In our final efforts to further reduce the space required for the index, the following two observations play an important role. Approximately 50% of the entries from all the conditional inverted lists in the index, have string-id corresponding to leaf node in Δ and have low score value (frequency of one). Moreover, the document array, which is a part of the triplet in the conditional inverted lists, does not contribute in the process of retrieving top-k answers and is used only during reporting to identify the documents with highest score. First observation suggests that pruning the conditional inverted list entries corresponding to leaf nodes would significantly reduce the index space. In particular, we do not store those triplets whose string-id field corresponds to a leaf node in Δ. The downside is that the modified index will no longer be able to report the documents with frequency of one. However, this shortcoming can be justified by reductions in space, and algorithmic approach can be employed to retrieve such documents if needed. Using second observation, we can choose to get rid of the document-id field and incur additional overhead during query time. In short, the document-id in the triplet corresponding to an internal node (string-id = pre-order rank of that node) is not stored explicitly in the conditional inverted lists. The string-id of a triplet in a conditional inverted list associated with a node u_i is replaced by a pointer pointing to triplet associated with the *highest-descendent* node in the subtree of u_i with the *same* document-id. Now the triplets in the conditional inverted lists are sorted according to the value of this pointers. Retrieval of the document-id can be done in an online fashion by chasing pointers from an internal node up to the leaf corresponding to that document. Though these modifications do not guarantee any theoretical bounds on query time ($O(n)$ in worst case), we observed that index-C performs well in practice.

4.4 Experimental Analysis

We evaluated our new index and its compressed variants for space and query time using english texts and protein collections. `ENRON` is a ≈100MB collection of 48619 email messages drawn from a dataset prepared by the CALO Project[1]. `PROTEIN` is a concatenation of 141264 Human and

[1] http://www.cs.cmu.edu/ enron/

Mouse protein sequences totaling ≈60MB[2]. We implemented all of the above indexes using the programming language C++, compiled with the g++ compiler version 4.2. Public code libraries [3] were used to develop some of the components in the indexes. Our experiments were run on an Intel Core 2 Duo 2.26GHz machine (MAC OS 10.6.5) with a 4GB RAM. In the following discussions, we first analyze the space-time tradeoffs for various indexes described in this chapter. Then we empirically compare these indexes with the inverted index when word boundaries are well defined and query patterns are aligned on word boundaries. Finally we evaluate the performance of our index for two pattern queries (with TF-IDF as a relevance metric) using heuristic algorithm.

4.4.1 Space-Time Tradeoffs

Figure 4.1 shows the space requirements for the original index and its compressed variants against input text size for both datasets. Reduction in the space requirements for index-B and index-C can be analyzed separately for the three key components of the indexes: document array, score array and string-id-array. Figure 4.2 shows the space utilization of these components for each of the proposed indexes.

For both document array and score array, even though it is possible to use the theoretically-minimal number of bits required per entry, it would result in a slowdown in the query time due to the lacking of efficient mechanisms for the retrieval of the array values. In index-B, recall that we try to keep the encoding simple and do not compress the data to the fullest extent so as to achieve reasonable compression and restrict query time within acceptable limit simultaneously. Particularly, as most of the values in the score (frequency) array ($\approx 97\%$ for `ENRON`, $\approx 98\%$ for `PROTEIN`) are less than 16, the proposed heuristic for compressing the score array in index-B achieves a very good practical performance. Out of three components, string-id array is the least compressible as its values correspond to the pre-order ranks of nodes in the suffix tree with ranges from 0 to $|T| = n$. We can utilize the fact that string-id array entries for a node are sorted in the increasing order by using difference encoding (such as gap) for efficient compression. However, such a method would

[2] http://www.ebi.ac.uk/swissprot

[3] http://www.uni-ulm.de/in/theo/research/sdsl.html, http://pizzachili.dcc.uchile.cl/indexes.html

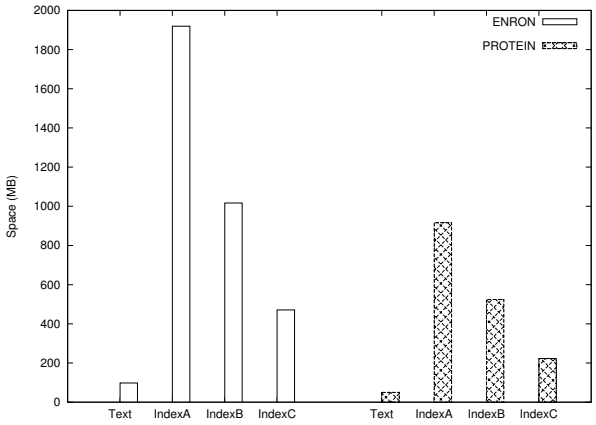

FIGURE 4.1. Space comparison of indexes

naturally incur a query time overhead. Instead, as mentioned in the previous section, index-B makes use of the run-length encoding to represent the consecutive entries with the same string-id value, and was able to eliminate $\approx 30\%$ string-id array entries for ENRON and $\approx 25\%$ string-id array entries for PROTEIN in our experiments. Using these compression techniques, index-B is ≈ 10 times the text as compared to index-A (≈ 20 times text).

Recall that index-C does not store the document id for each entry explicitly to achieve space savings, at the expense of a slightly longer time to report the documents. Space savings are also achieved when we prune the inverted list entries corresponding to the leaf nodes, which account for 50% in ENRON and 55% in PROTEIN of the total number of entries. As a result, index-C improves further on index-B and takes only ≈ 5 times of the text in the space requirement.

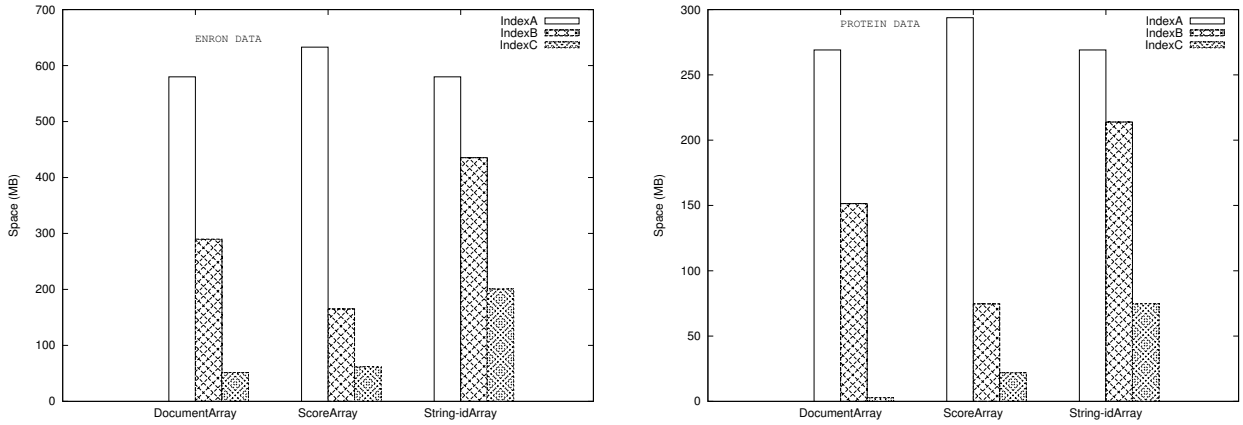

FIGURE 4.2. Compression achieved for each of three components in Conditional Inverted Lists

For these experiments, 250 queries from ENRON and 125 queries from PROTEIN, which appear in at least 10 documents with frequency 2 or more, are generated randomly for pattern lengths varying from 3 to 10. This therefore forms a total of 2000 and 1000 sample queries for ENRON and PROTEIN, respectively. In addition, we ensure that the selected patterns of length 3 appear in at least 80 documents to observe the practical time in reporting top-k (k = 10, 20, ..., 80) documents. Figure 4.3 shows the average time required to retrieve $k = 10$ documents with the highest score (frequency) for patterns with varying lengths. Average time required for retrieving documents in descending order of score (frequency) for a set of patterns with length 3 is shown in Figure 4.4 for varying k. These figures show that space savings achieved by the successive variants of our index (with increasing level of compression) will not hurt the query time to a great extent. A nearly linear dependance of query time on pattern length and k can also be observed from these figures. Matching the pattern P in compressed suffix tree Δ and binary search to obtain intervals in conditional inverted list of nodes in compressed suffix tree during top-k retrieval dominates the query time for index-A. Occasional slight drop in the query time for the indexes for increasing pattern length can be attributed to the binary search as it depends on the number of documents in which the query pattern is present. Query timings for index-B closely follow to that of index-A, with decoding the score (frequency) values for possible top-k candidates being primarily responsible for the difference. Index-C has an additional overhead of decoding the document-id for each top-k

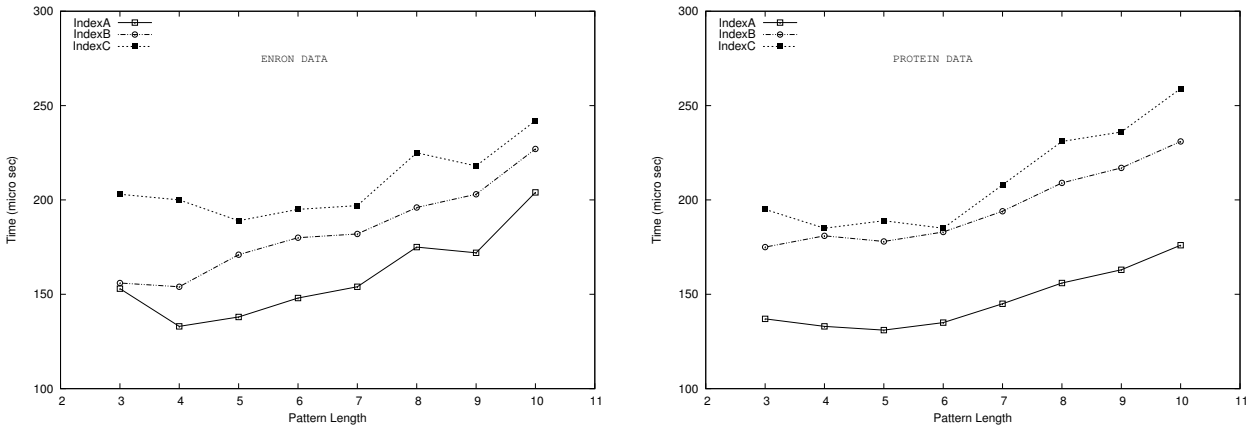

FIGURE 4.3. Effect of pattern length ($k = 3$)

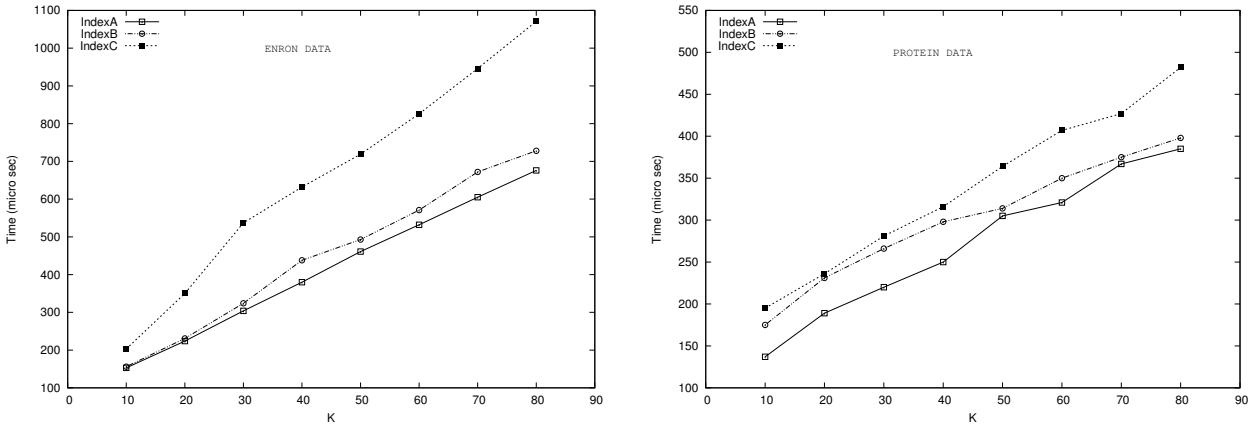

FIGURE 4.4. Effect of k ($|P| = 3$)

answer to be reported. As a result, the gap in the query time of index-C with the other indexes should gradually increase with k, as is observed in the Figure 4.4.

4.4.2 Word/Term Based Search

In this subsection, we compare our phrase indexes with the traditional inverted index, highlighting the advantages of the former ones over the latter. For a fair comparison, our proposed indexes in this subsection are built on the word suffix tree instead of the generalized suffix tree (Theorem 4.8) so as to support searching of only those patterns that are aligned with the word boundaries. We begin by comparing the query times. Traditional inverted index are known to be efficient for single-word searching. When the inverted lists are each sorted in descending order of score, ranked retrieval of documents would simply return the initial entries from the list corresponding to the query word. However, for efficient phrase searching, sorting the document lists by document-id (instead of score) would allow faster intersections of multiple lists. Figure 4.5 shows the time required for retrieving top-10 documents with highest score (frequency) for a set of phrases consisting of two and three words, respectively. Here, we generated 800 additional queries aligned on english word boundaries from ENRON. Traditional inverted index has its inverted lists sorted according to the document ids as mentioned, and we apply finger binary search [69] for intersecting multiple lists. We do not report the results when inverted lists are sorted by score as the timings were significantly worse. Figure 4.5 show that our phrase indexes perform much better than the intersection-based retrieval, and the performance degradation in traditional inverted index would become more serious with the

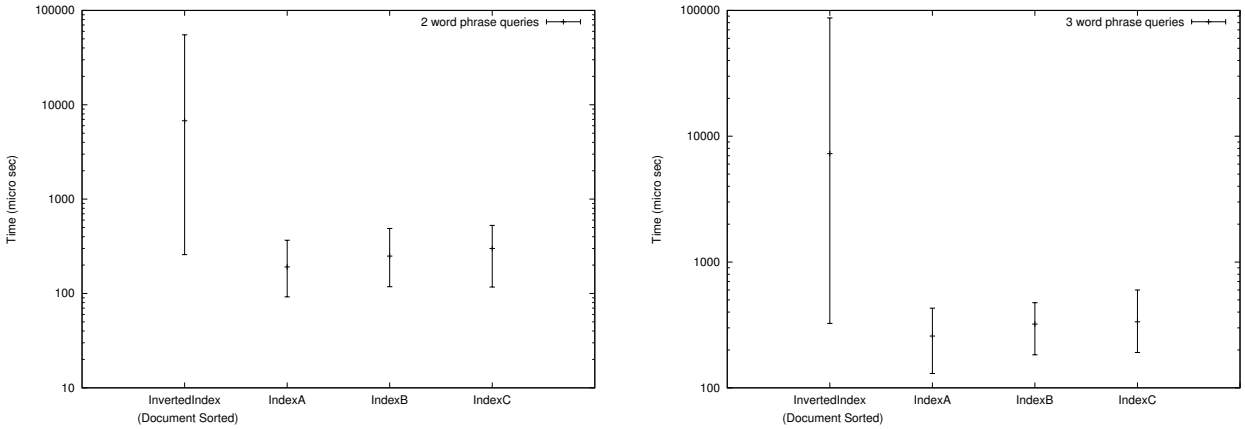

FIGURE 4.5. Time (high, low, mean) for a set of phrase queries ($k = 10$)

increase in words in a phrase query. Query times of our string/phrase indexes show that its query time for reporting top-10 documents is in the range of 100-400 microseconds, thus achieving good practical performance.

A key point behind the widespread usage of the inverted index is that it can be stored in little space when compared with the size of input document collection; 20%-60% or more depending on whether it includes the position lists. One way to avoid the intersection of position lists in the phrase queries would be to store inverted list of all phrases up to some fixed number (say h) of words. Such an index still has to reply on intersection for phrases with more than h words. Figure 4.6 shows the space requirement for this variant of inverted index without the position lists. From the figure, it is clear that the space required for such a solution gradually increases with h and directly depends on the number of distinct phrases in the input text. In contrast, our phrase index supports phrase

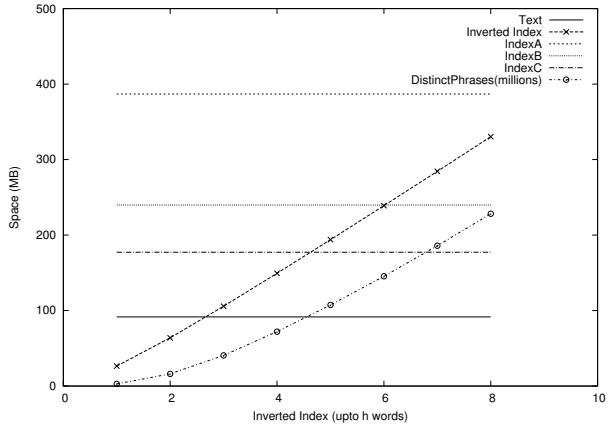

FIGURE 4.6. Space for inverted index

searching with arbitrary number of words. In the most space-efficient version of our phrase index (index-C), it takes just under two times of the input text in space. With gradual increase in space required, the traditional inverted index for phrases up to h words occupies more space than index-C for all $h \geq 5$. It is important to note that the traditional inverted index is maintained as an additional data structure along with the original text, whereas our proposed indexes are self indexes and do not need original text. Thus, our phrase index compares favorably against the traditional inverted index for phrase searching in practice.

4.5 Top-k TF-IDF Queries

In web search engines, *tf-idf* (term frequency–inverse document frequency) [9] is one of the most popular metric for relevance ranking. The query consists of multiple keywords (patterns), say $P_1, P_2, ..., P_m$ and the score of a document d, $score(d)$, is given by $score(d) = \sum_{i=1}^{m} tf(P_i, d) \times idf(P_i)$, where $tf(P_i, d)$ denotes the number of occurrences of P_i in d, and $idf(P_i) = \log \frac{|\mathcal{D}|}{1+docc(P_i)}$, with $|\mathcal{D}|$ representing the total number of documents and $docc(P_i)$ representing the number of documents containing pattern P_i. Many other versions of this metric are available in the literature. For top-k document retrieval that is based on the *tf-idf* metric (with multiple query patterns), most of the existing solutions are based on heuristics. When the query consists of a single pattern, the inverted index with document lists sorted in score order can retrieve top-k documents in optimal time. However, for an m-pattern query (a query consisting of m patterns say $P_1, P_2, ..., P_m$), we may need the inverted lists sorted according to the document id as well. In this section, we introduce an exact algorithm and compare the results obtained by applying it to inverted index as well as our index (index-B). Although our algorithm does not guarantee any worst-case query bounds, the focus is to explore the capabilities of our index as a generalized inverted index. Along with our index, we make use of a wavelet tree [55] over the document array for its advantages in offering dual-sorting functionalities. We restrict the query patterns to words in order to give a fair comparison between our index and the inverted index.

Suppose that N denotes the number of suffixes in the word suffix tree. Let $DA[1...N]$ be an array of document ids, such that $DA[i]$ is the document id corresponding to ith smallest suffix

(lexicographically) in the word suffix tree. Note that each entry in DA takes at most $\lceil \log |D| \rceil$ bits to store. Therefore a wavelet tree *W-Tree* of DA can be maintained in $N \log |\mathcal{D}|(1 + o(1))$ bits. Now, given the suffix range $[\ell, r]$ of any pattern P, the term frequency $tf(P, d_j)$ for the document with id j can be computed by counting the number of entries in DA with $DA[i] = j$ and $\ell \leq i \leq r$. This query can be answered in $O(\log |\mathcal{D}|)$ time by exploring the orthogonal range searching functionality of *W-Tree*. Since term frequency in any document can be computed using *W-Tree*, we do not store the score (term frequency) array in index-B. This slightly compensates for the additional space overhead due to *W-Tree*. Inverse document frequency idf can be computed using Theorem 3. For simplicity, we describe the algorithm for two pattern queries (P_1 and P_2) as follows, and the algorithm can be easily extended for the general m-pattern queries. Let S_{ans} and S_{doc} be two sets of documents which are set to empty initially, and let d_1^k and d_2^k represents the kth highest scoring document corresponding P_1 and P_2, with *term frequency* as the score function and $score(d) = tf(P_1, d)\ idf(P_1) + tf(P_2, d)\ idf(P_2)$.

Algorithm 1 Answering top-k *tf-idf* query involving two patterns

$S_{ans} = S_{doc} = \{\}, x = y = 1$
while $|S_{ans}| < k$ **do**
  if $score(d_1^x) \geq score(d_2^y)$ **then**
    $S_{doc} \leftarrow S_{doc} \cup d_1^x$ and $x \leftarrow x + 1$
  else
    $S_{doc} \leftarrow S_{doc} \cup d_2^y$ and $y \leftarrow y + 1$
  end if
  if $|S_{doc}| = 1, 2, 4, 8, 16, ...$ **then**
    $score_{max} = tf(P_1, d_1^x)\ idf(P_1) + tf(P_2, d_2^y)\ idf(P_2)$
    for each $d \in S_{doc}$ **do**
      if $score(d) \geq score_{max}$ and $d \notin S_{ans}$ **then**
        $S_{ans} \leftarrow S_{ans} \cup d$
      end if
    end for
  end if
end while
Choose k documents in S_{ans} with the highest $score$ value

The main idea of the algorithm is to maintain a list of candidate top-k documents in the set S_{doc}, and refine the candidate set by moving documents to the set S_{ans} from time to time. Each document

in S_{ans} will have score higher than an imaginary $score_{max}$, and the set S_{ans} will always contain the highest scoring documents we have examined so far. The algorithm stops as soon as S_{ans} contains k documents, in which we report the top-k documents from the set.

Experimental analysis: We compare the performance of our index against the traditional inverted index for answering 2-pattern queries using the algorithm as described above. In the traditional inverted index, document lists are sorted either by score (frequency) or document-id. To apply the above heuristic, we need dual-sorted documents lists, where each list is sorted on both score as well as document-id. Score sorted lists support ranked retrieval of documents for individual patterns but *tf-idf* score can not be computed efficiently. If lists are sorted by document-id, though *tf-idf* score computation is faster, document retrieval in ranked order is not efficient. As a result we first duplicate the document lists for each of the pattern P_i and sort them as required. Figure 4.7 shows the mean time required for retrieving top-k documents for a set of fifty 2-pattern queries for ENRON such that each pattern is highly frequent. As observed from the figure, query time for our index increases faster than that of the inverted index.

We remark that the major part of the query time used by the inverted index is on re-sorting the the document lists in which the query patterns occur. Thus, if the patterns are not too frequently occurring, the time spent on re-sorting is reduced, and the advantages of our index over the inverted index will vanish. Finally, the size of our index is ≈ 3.1 times of the text size.

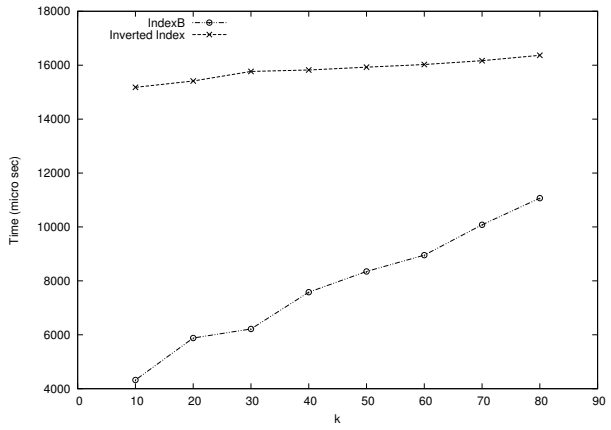

FIGURE 4.7. Answering 2-pattern queries

4.6 Related Work

Suffix trees and suffix arrays are efficient data structures which can be used to index a text and support searching for any arbitrary pattern. These data structures can be maintained in linear space and can report all the occurrence of a pattern P in optimal (or nearly optimal) time. The space-efficient versions of suffix trees and suffix arrays are called compressed suffix trees and compressed suffix arrays, respectively, which take space close to the size of the indexed text. From a collection $\mathcal{D}$ of $|\mathcal{D}|$ documents $\{d_1, d_2, ..., d_{|\mathcal{D}|}\}$ of total length n, the problem of reporting documents containing a query pattern P is called the "document listing" problem. This problem was first studied by Matias et al. [94], where they proposed a linear space index with $O(p \log n + |output|)$ query time; here, p denotes the length of the input pattern P and $|output|$ denotes the number of the qualified documents in the output. An index with optimal $O(p + |output|)$ query time was later achieved in [99]. Sadakane [119] showed how to solve the document listing problem using succinct data structures, which take space very close to that of the compressed text. He demonstrated how to compute the *tf-idf* [9] of each document with the proposed data structures. Similar work was also done by Välimäki and Mäkinen [129] where the authors derived alternative succinct data structures for the problem.

In many practical situations, we may be interested in only a few documents which are highly relevant to the query. Relevance ranking refers to the ranking of the documents in some order, so that the result returned first is what the user is most interested in. This can be the document where the given query pattern occurs most number of times (frequency). The relevance can also be defined by a similarity metric, such as the proximity of the query pattern to a certain word or to another pattern. This problem is modeled as top-k document retrieval, where the task is to retrieve the k highest scoring documents based on some score function. An $O(n \log n)$ words index has been proposed in [62] with $O(p + \log |\mathcal{D}| \log \log |\mathcal{D}| + k)$ query time. Hon et al. [66] proposed a linear-space index ($O(n)$ words) with nearly optimal $O(p + k \log k)$ query time. Yet, the constants hidden in the space bound restricts its use in practice. Culpepper et al. [33] proposed a space-efficient practical index

based on wavelet trees [55], but their query algorithm is based on a heuristic, so that it does not guarantee any worst-case query performance.

The most popular ranking function in web search applications is *tf-idf* [9]. Under the *tf-idf* model, Persin et al. [114] give different heuristics to support top-k ranked retrieval when the inverted lists are sorted in decreasing order of the *tf* score. Various generalizations of this are studied by Anh and Moffat [6] under the name "impact ordering". In [103], Navarro and Puglisi showed that wavelet trees can be used for maintaining dual-sorted inverted lists corresponding to a word, where the documents can efficiently be retrieved in score order or in document id order. Recently, Hon et al. [63] proposed an index for answering top-k multi-pattern queries. On a related note, top-k color query problems (with applications in document retrieval) have been studied in [49, 81].

4.7 Summary

This chapter introduces the first practical version of inverted index for string documents. The idea is to store lists for a selected collection of substrings (or phrases) in a conditionally sorted manner. Succinct data structures are used to represent these lists so as to reap benefits of dual sorting and achieve good top-k retrieval performance. We show how top-k *tf-idf* based queries can be executed efficiently. Furthermore, our indexes show a space-time advantage over all of the traditional techniques for searching long phrases. With this being the first prototype, more research in the area has helped in deriving structures with high practical impact.

Chapter 5
Categorical Range Maxima Queries

5.1 Introduction

Given an array A of n elements from a totally ordered set, a natural question is to ask for the position of a maximum element between two specified indices a and b. Queries of this form are known as range maximum queries (RMQ). Consider a sample query: "Give me the highest paid employee within age group 18 to 22 years". By arranging all employees in a age-sorted array with his/her salary as the key, this query translates into an RMQ problem. Being an important tool in designing data structures for numerous problems in string processing and computation geometry, RMQ has been extensively studied in the literature [16, 119, 15, 45]. There are several variants of the problem, the most prominent being the one where the array is static and known in advance. The current best known result for such a scenario is by Fischer and Heun [45], where they present a $2n + o(n)$-bit structure capable of answering queries in constant time.

However, in many applications, the standard RMQ problem does not suffice. Consider the generalization of the above query as a motivating example: "Give me the list of highest paid employees for different job positions (one per job position) with age between 18 to 22 years". This problem can obviously be solved by maintaining age-sorted array of employees as before for each designation in the organizational hierarchy and then issuing a RMQ for all of them. However, this solution may be very inefficient as the job positions held by employees within the specified age group can be only a fraction of all listed positions for the organization. We call the above problem to be an instance of *Categorical Range Maxima Query* (CRMQ). For CRMQ, we assume that each element in the input array A is assigned a color. The goal is to preprocess the array and maintain a data structure, such that given a query range $[a, b]$, one can efficiently report each distinct color c in the query range along with the highest element in $A[a...b]$ with color c. Further continuing the example under consideration, lets say we only need to output the job positions where the highest

paid employee with that designation earns more than \$80,000 per year. This natural extension of CRMQ called "threshold-CRMQ" problem is formally defined below.

Problem 1. [Threshold-CRMQ] Let $A[1...n]$ be an array of n distinct integers in $[1, n]$ with each element $A[i]$ associated with a color $C[i] \in [\sigma]$. Then goal is to build a data structure such that, given a query (a, b, τ), we can report the triplet $(c, p_c, A[p_c])$ for those colors $c \in [\sigma]$ with $A[p_c] \geq \tau$. Here $A[p_c]$ represents the highest element in $A[a...b]$ with color c. If there does not exist an element in $A[a...b]$ with color c, then $A[p_c] = -\infty$.

Top-k queries are widely popular in database and information retrieval systems as they allow users to focus on the most important k outputs amongst those which satisfy the query. We also study top-k version of CRMQ problem (top-CRMQ), where the query input consists of a range $[a, b]$ and an integer $k \leq \sigma$, and we are required to output only k colors with the highest $A[p_c]$ values.

Problem 2. [Top-CRMQ] Let $A[1...n]$ be an array of n distinct integers in $[1, n]$ with each element $A[i]$ associated with a color $C[i] \in [\sigma]$. Then goal is to build a data structure such that, given a query (a, b, k), we can report k triplets $(c, p_c, A[p_c])$ for colors $c \in [\sigma]$ with the highest $A[p_c]$ values, where $A[p_c]$ represents the highest element in $A[a...b]$ with color c. If there does not exist an element in $A[a...b]$ with color c, then $A[p_c] = -\infty$.

In this article, we focus on top-CRMQ as our central problem. We distinguish between the sorted and unsorted version of this problem. In the sorted version, a triplet $(c, p_c, A[p_c])$ is reported before $(c', p_{c'}, A[p_{c'}])$, if $A[p_c] > A[p_{c'}]$, whereas unsorted version do not place any such restrictions. We focus on sorted version in RAM model and unsorted version in external memory. For the rest this paper, we use the following notations: $\log^{(1)}(\cdot) = \log(\cdot), \log^{(h)}(\cdot) = \log(\log^{(h-1)}(\cdot))$ for $h \geq 2$, and $\log^*(\cdot)$ is the minimum h such that $\log^{(h)}(\cdot) \leq 2$. Our main results are summarized in following theorems.

Theorem 5.1. There exists a linear space (in words) and optimal $O(k)$ time solution for the (sorted) top-CRMQ problem in RAM model.

Theorem 5.2. There exists an external memory structure of $O(n \log^* n)$ space and optimal $O(1+\frac{k}{B})$ query I/Os for the top-CRMQ problem, where $\log^* n$ is the iterated logarithm of n and B is the block size.

Theorem 5.3. There exists an external memory structure of linear-space and near-optimal $O(\log^* n+ \frac{k}{B})$ query I/Os for the top-CRMQ problem, where $\log^* n$ is the iterated logarithm of n and B is the block size.

Answering threshold-CRMQ: Data structures for answering top-CRMQ as summarized in theorems above, can be used for answering the threshold-CRMQ as well. Given a threshold-CRMQ (a, b, τ), we issue multiple top-CRMQ's as follows. Assume, we are using the I/O-optimal structure in Theorem 5.2, then we choose $K_j = 2^j B$ and issue top-CRMQ (a, b, K_j) for $j = 0, 1, 2, 3, ...$ until we find the smallest K_j (say K') where at least one of the triplet $(x, p_x, A[p_x])$ in the output set violates the condition $A[p_x] \geq \tau$. Then all those triplets corresponding to the output of top-CRMQ (a, b, K_j) satisfying the condition $A[\cdot] \geq \tau$ can be reported as the final answers. The number of I/O's required is $O(1 + 2 + 4 + ... + K'/B) = O(1 + K'/B) = O(1 + k/B)$, where k is the output size. If we are using the linear-space structure, we use the same procedure, with $K_j = 2^j B \log^* n$ and the query I/Os can be bounded by $O(\log^* n + (1 + 2 + 4 + ... + k/B)) = O(\log^* n + k/B)$. In conclusion, results in Theorem 5.2 and Theorem 5.3 are applicable for threshold-CRMQ as well.

5.2 Applications of CRMQ

5.2.1 Sorted Dominance Reporting

In this problem, we want to store an array A in a data structure such that for any query range $[a, b]$ all elements $A[i]$, $a \leq i \leq b$, can be reported in sorted order. Brodal et al. [19] described a linear space data structure that answers such queries in $O(b - a + 1)$ time, moreover, their data structure can also be used to report k highest points in the range in sorted order. Karpinski and Nekrich [81] considered the same problem in the color scenario: elements of the array are also assigned colors. We assume that colors are from an ordered set; now the query answer must report the k highest colors that occur in the query range and colors must be reported in the reverse order. We observe

that the optimal data structure described in Theorem 5.1 generalizes the result of [81, 19]. This result is obtained using a new data structure for sorted three-dimensional dominance queries, which may be of independent interest. The result is summarized below (Proof is deferred to Section 5.8).

Theorem 5.4. A given set of n three-dimensional points can be maintained as an $O(n)$-word data structure that can answer a three-dimensional dominance reporting query in $O(\log n + |output|)$ time in RAM model, with outputs reported in the sorted order of z coordinate.

5.2.2 Ranked Document Retrieval

Suppose that we want to store a collection $\mathcal{D} = \{d_1, d_2, ..., d_D\}$ of D documents (strings) of total n characters, so that for a given query string P all documents containing P can be reported. This problem can be reduced to one-dimensional color reporting problem and can be solved optimally [99]. A more general and arguably the most important query, known as the *top-k document retrieval query* asks to find those k documents in $\mathcal{D}$ which are most relevant to P, where k is also an input parameter. The relevance of a document d w.r.t a pattern P is captured using a predefined ranking function $w(P, d)$, which is dependent on the set of occurrences of P in d. A popular example is the *term frequency*, where $w(P, d)$ is the number of occurrences of P in d. This problem has been studied extensively in string searching community (See [101] for an excellent survey) and linear-space and optimal query time internal memory results are known [66, 102]. Whereas in external memory, the best known linear space index is given by Shah et al. [122], however the query I/O bound is $O(\frac{|P|}{B} + \log_B n + \log^{(h)} n + \frac{k}{B})$ I/Os for any constant $h \geq 1$. We show that our solution for top-CRMQ can be used to obtain the following new result.

Theorem 5.5. If the ranking function is such that, the relevance of a document w.r.t. a pattern is not more that its relevance w.r.t. to any prefix of the same pattern, then we can construct a linear-space structure for answering top-k document retrieval queries in $O(\frac{|P|}{B} + \log_B n + \log^* B + \frac{k}{B})$ I/Os, where P is the input pattern.

To derive the index summarized in above theorem, first construct a generalized suffix tree [132] of the document collection. Then we mark nodes with document-ids as follows: a leaf node ℓ is

marked with document d_j if the suffix represented by ℓ belongs to d_j. An internal node u is marked with d_j if it is the lowest common ancestor of two leaves marked with d_j. Notice that a node can be marked with multiple documents. For each node u (with pre-order rank $rank(u)$) and each of its marked documents d_j, we define a triplet $(rank(u), w(path(u), d_j), d_j)$, where $path(u)$ represents the concatenation of edge labels on the path from root to u. Let (x_i, y_i, d_{c_i}) represents the i-th triplet, where $x_i \leq x_{i+1}$, then we construct A and C as follows: $A[i] = y_i$ and $C[i] = c_i \in [1, D]$. The top-k documents corresponding to the query (P, k) are same as the output colors for top-CRMQ (a, b, k), where $[a, b]$ represents the maximal range such that for all triplets $(x_i, \cdot, \cdot)$ with $i \in [a, b]$, the node with pre-order rank x_i is in the subtree of u_P. Here u_P represents the locus of P, the node closest to root with P as a prefix of $path(u_P)$. Using a String B-tree [43] and some auxiliary structures occupying $O(n)$-word space over all, we can compute u_P in $O(\log_B n + \frac{|P|}{B})$ I/Os. We note that, this result require relevance to be a monotonic function.

5.2.3 Categorical Range Reporting Without Duplicates

In the categorical (or colored) range reporting problem the set of input points is partitioned into categories and stored in a data structure; a query asks for categories of points that belong to the query range. The problem has been extensively studied in computational geometry and database communities [75, 60, 18, 99, 81, 104, 85, 86].

In three-sided color reporting, the query asks to report the set of colors of the points in an input region $[a, b] \times [\tau, +\infty)$. Without loss of generality, we assume the points are in rank-space. [1] The first external memory result for this problem was given by Nekrich [104]. His results on this problem were further improved by Larsen and Walderveen [86], where they presented an $O(nh)$-word data structure with $O(\log^{(h)} n + \frac{k}{B})$ query cost, k being the output size, $1 \leq h \leq \log^* n, \log^{(h)} n = \log\log^{(h-1)} n$ and $\log^{(1)} n = \log n$. Thus, by choosing $h = \log^* n$, an I/O-optimal structure can be obtained. On the other-hand, a linear space structure can be obtained by choosing $h = O(1)$.

[1] By rank-space we assume the points are in $[n] \times [n]$ grid, and the projections of any two points to either axis is different. If the points are in a $[U] \times [U]$ grid, we can reduce them to $[n] \times [n]$ grid using standard techniques. However the space will increase by an $O(n)$ words and the query cost by $O(\log\log_B U)$ I/Os (or $O(\log\log U)$ time). If the coordinate values are unbounded, the extra term in space is again $O(n)$, but in the query cost is $O(\log_B n)$ I/Os (or $O(\log n)$ time).

The data structures described in [104, 86] have a limitation that can compromise their usefulness in certain situations: the list of colors in the output set may contain several (yet constant) occurrences of the same color. Eliminating such duplicates (in the current settings) needs extra I/Os (sorting is inevitable in these solution, which makes these results less-optimal in terms of query I/Os). In [104], another data structure that uses linear space and reports every color exactly once is described. Unfortunately, this data structure needs $O((\frac{n}{B})^{\varepsilon} + \frac{k}{B})$ I/Os to answer a query, where ε is an arbitrarily small positive constant. We provide the solution for this important open problem that requires every color to be reported exactly once.

Theorem 5.6. A three-sided color reporting query on a set of n points in rank-space can be answered in $O(1+\frac{k}{B})$ I/Os using an $O(n \log^* n)$-word structure, or in $O(\log^* n+\frac{k}{B})$ I/Os using an $O(n)$-word structure, such that the output set contains *exactly one copy of each answer*, where k is the output size, $\log^* n$ is the iterated logarithm of n and B is the block size.

Proof. Let $P = \{(i, y_i)|i = 1, 2, 3, ..., n\}$ be the set of points, then construct the array A, where $A[i] = y_i$ and its color is same as that of (i, y_i). Then the output of any three-sided color reporting query on P with $[a, b] \times [\tau, n]$ as an input is the same as that of a threshold-CRMQ (a, b, τ) on A. Thus, we obtain the results summarized in the above theorem using Theorem 5.2 and 5.3. □

Consequently, we achieve a smaller non-optimal term of $\log^* n$ in the I/O bound of the linear-space structure compared to the $(\frac{n}{B})^{\varepsilon}$ or $\log^{(O(1))} n$ terms in the existing solutions. Moreover, using standard techniques [104, 86] in conjunction with results in Theorem 5.2, Theorem 5.3, we obtain following results for (two dimensional) four-sided color reporting problem. Although this improves the known results of the problem [86], the output set may contain multiple (at most twice) copies of the same color.

Theorem 5.7. A four-sided color reporting query on a set of n points in an $[n] \times [n]$ grid can be answered in $O(1 + \frac{k}{B})$ I/Os using an $O(n \log n \log^* n)$-word structure, or in $O(\log^* n + \frac{k}{B})$ I/Os using an $O(n \log n)$-word structure. Here k is the output size, $\log^* n$ is the iterated logarithm of n and B is the block size.

5.3 Top-k to Threshold Mapping

Before moving on to the proposed data structure and query answering, we prove the following result in this section: using a linear space data structure of $O(n)$-word, we can compute a threshold $\tau^k_{a,b}$ for a given top-CRMQ (a, b, k) in $O(1)$ time such that size of $Out_t = \{c \in \Sigma, A[p_c] \geq \tau^k_{a,b}\}$ is bounded by $k + O(k)$, where $A[p_c]$ represents the highest element in $A[a...b]$ with color c

Data structure: We partition the array $A[1...n]$ into $\lceil \frac{n}{\log^2 n} \rceil$ disjoint blocks each of size $\log^2 n$ (possibly except for the rightmost block). Starting from each blocking boundary, we consider spans (of length at most n) covering $1, 2, 4, 8, ...$ blocks, and for each such span $S = A[x...y]$, we maintain $\tau^k_{x,y}$ for $k = 1, 2, 4, 8, ..., n$. Here $\tau^k_{x,y} \in \{A[j]|j \in [x, y]\}$ with k as the output size of the threshold-CRMQ $(x, y, \tau^k_{x,y})$. This takes linear space i.e., $O(n)$ words. Further, we divide each block into sub-blocks of size $\log^2 \log n$, and starting from each sub-block boundary, we consider spans (of length at most $\log^2 n$) covering $1, 2, 4, 8, ...$ sub-blocks. Again, for each such span $S' = A[x'...y']$, we maintain $\tau^k_{x',y'}$ for $k = 1, 2, 4, 8, ..., \Theta(\frac{\log^2 n}{\log^2 \log n})$. Notice that the explicit storage of $\tau^k_{x',y'}$'s (using $\log n$ bits per element) is costly. Therefore, we simply encode its relative position within that span in $O(\log(\log^2 n)) = O(\log \log n)$ bits occupying $O(n)$ words space overall. Finally, answers for the query (a, b, k) where both the inputs a, b are completely within a sub-block can be maintained in $o(n)$ bits using tables.

Query answering: In order to compute the threshold $\tau^k_{a,b}$ corresponding to the input (a, b, k), we get k' by approximating k to the next highest power of 2 i.e., $k' = 2^{\lceil \log k \rceil}$. Then the input range $[a, b]$ can be partitioned into (at most) 6 spans $[a, a' - 1], [a', a'' - 1], [a'', b''], [b'' + 1, b'], [b' + 1, b]$ such that (1) both $[a, a' - 1]$, $[b' + 1, b]$ are within a sub-block, (2) $[a', a'' - 1]$, $[b'' + 1, b']$ are covered by spans of sub-blocks and (3) $[a'', b'']$ is covered by two possibly overlapping spans of blocks. The $\tau^{k'}_{\{\cdot,\cdot\}}$ for each of these spans can be retrieved in constant time and we choose the maximum among them as our threshold $\tau^k_{a,b}$. It can be easily verified that $\hat{k} \leq 6k' < 12k$ and $\hat{k} \geq min(k, dcol)$, where $dcol$ denotes the number of distinct colors in $C[a...b]$.

5.4 The Framework

For color listing problem i.e., to simply enumerate all distinct colors in $C[a...b]$, Muthukrishnan [99] proposed the *chaining* idea, where each occurrence of a particular color points to (or chains to) its predecessor of the same color [2]. Therefore, among all occurrences of a particular color $c \in [\sigma]$ occurring in $C[a...b]$, only the first ones chain will be pointing outside the range $[a, b]$. Based on this observation, he reduced the problem to a (two-dimensional) three-sided range reporting query, which can be solved optimally using known structures. We introduce a generalization of this approach for solving our top-CRMQ problem. Formally, for each position $i \in [1, n]$ in the array A, we define *previous* and *next* pointers as follows:

$$prev(i) = max\{\{j \in [1, i) | A[j] > A[i], C[j] = C[i]\} \cup \{-\infty\}\}$$

$$next(i) = min\{\{j \in (i, n] | A[j] > A[i], C[j] = C[i]\} \cup \{+\infty\}\}$$

Using these pointers, for each position $i \in [1, n]$ in A we obtain a (weighted) interval-pair with $(prev(i), i)$ as a backward interval, $(i, next(i))$ as a forward interval, and $A[i]$, $C[i]$ being the weight and color associated with the interval-pair respectively. We represent such an interval-pair by a pentuple $(i, A[i], C[i], prev(i), next(i))$. The following is a key observation for the *two-sided chaining* just introduced.

Lemma 5.8. For a given range $[a, b]$ and a color c, let $S_{a,b,c} = \{i_1, i_2, ..., i_r\}$ be the (possibly empty) set of all positions within $[a, b]$ such that $C[i_1] = C[i_2] = ... = C[i_r] = c$. If $S_{a,b,c}$ is not an empty set, then exactly one element $p_c \in S_{a,b,c}$ satisfies the following: $prev(p_c) < a, b < next(p_c)$, where $A[p_c] = max\{A[i_1], A[i_2], ..., A[i_r]\}$.

In order to utilize the above lemma for answering top-CRMQ, we use an $O(n)$-word structure that can compute a threshold $\tau^k_{a,b}$ for a given top-CRMQ (a, b, k) in $O(1)$ time such that size of $Out_t = \{c \in \Sigma, A[p_c] \geq \tau^k_{a,b}\}$ is bounded by $\hat{k} = k + O(k)$, where $A[p_c]$ represents the highest element in $A[a...b]$ with color c (see Section 5.3). Then, Lemma 5.8 suggests that if a triplet

[2] If there is no such predecessor, then points to $-\infty$.

$(c, p_c, A[p_c])$ is an answer for a top-CRMQ, then the pentuple $(p_c, A[p_c], C[p_c], prev(p_c), next(p_c))$ satisfies the following conditions, and vice versa: $p_c \in [a, b]$, $prev(p_c) < a$, $next(p_c) > b$ and $A[p_c] \geq \tau^k_{a,b}$. Therefore, top-CRMQ can be reduced to a new problem as defined below.

Problem 3. Maintain a set $\mathcal{I}$ of n interval-pairs of the form $(i, A[i], C[i], prev(i), next(i))$ as a data structure, such that given a query $(a, b, k, \tau^k_{a,b})$, we can efficiently report all those interval-pairs with weight $\geq \tau^k_{a,b}$ and its backward, forward intervals stabbed by a, b respectively. i.e., output the interval-pairs satisfying the following five constraints:

(1) $prev(i) < a$ (2) $a \leq i$ (3) $i \leq b$ (4) $b < next(i)$ (5) $A[i] \geq \tau^k_{a,b}$

Notice that the output set Out_t for the above problem, is a super set of the output set Out_k of our top-CRMQ, because $\hat{k} \geq k$. Therefore, in order to answer a top-CRMQ, we first find the triplet $(c^*, p_{c^*}, A[p_{c^*}]) \in Out_t$ using a selection algorithm such that the number of triplets $(c, p_c, A[p_c]) \in Out_t$ with $A[p_{c^*}] \leq A[p_c]$ is k. This takes only $O(\hat{k}/B) = O(k/B)$ I/Os [17, 123]. Then, all those triplets in Out_t with $A[p_{c^*}] \leq A[p_c]$ can be reported as the final outputs. Both the problems being equivalent, we use the term "top-CRMQ" to refer to either of these problems. In particular, by top-CRMQ (a, b, k) we refer to Problem 2 whereas by top-CRMQ $(a, b, k, \tau^k_{a,b})$ we refer to the Problem 3. Moreover, for notational simplicity, input to the Problem 3 is defined as a quadruple (a, b, k, τ).

5.5 Interval Tree Based Solution

In this section, we present a simple interval-tree based external memory data structure and achieve the result summarized in the following lemma.

Lemma 5.9. A given set $\mathcal{I}$ of interval-pairs can be maintained as an $O(|\mathcal{I}|)$-space structure such that given a top-CRMQ (a, b, k, τ), all interval-pairs $(i, A[i], C[i], prev(i), next(i)) \in \mathcal{I}$ with $i \in [a, b]$, $prev(i) < a$, $next(i) > b$ and $A[i] \geq \tau$ can be reported in $O(\log^3(|\mathcal{I}|/B) + \frac{k}{B})$ I/Os.

We begin by describing a linear space external memory interval tree (which is not optimal, but is sufficient for our purpose) and then use it to answer top-CRMQ in the following subsections.

5.5.1 Linear Space Interval Tree

Given a set $\mathcal{I}$ of n intervals of the form (s_i, e_i), where s_i and e_i represent the start and end points, the output of an interval stabbing query is the set of intervals stabbed by a input point q; i.e., we need to output all those intervals (s_j, e_j) such that $q \in [s_j, e_j]$. For simplicity we assume all start and end points to be distinct; otherwise ties can be broken arbitrarily.

The proposed interval tree construction beings with building a balanced binary search tree (BST) of n nodes over all end points e_i of set $\mathcal{I}$. Thus, each node u in BST is associated with a unique end point which we denote as $stab(u)$[7]. Further, each node u is associated with a set of intervals $\mathcal{I}(u) = \{(s_i, e_i) | stab(u) \in [s_i, e_i], stab(v) \notin [s_i, e_i]$, where v is any ancestor of $u\}$. Let $size(u)$ represent the number of leaves in the subtree of u. We finish the construction by making each node u with $size(u) \leq B$, $size(parent(u)) > B$, a leaf node by first setting $\mathcal{I}(u) = \cup_{v \in subtree(u)} \mathcal{I}(v)$ and then pruning its subtree. We emphasize that, in this interval tree, for each leaf u, $\mathcal{I}(u)$ is bounded by $O(B)$[8]. The size of interval tree can now be bounded as $O(n)$ words since $\sum_u |\mathcal{I}(u)| = |\mathcal{I}| = n$. To answer a stabbing query, we first identify the node u_q such that value $stab(u_q)$ is the predecessor of q. Then any interval stabbed by a query point q will be associated with one of the $O(\log(\frac{n}{B}))$ nodes on the path from the root to node u_q. We summarize this property in the following lemma.

Lemma 5.10. Given a query point q, we can obtain a set of $O(\log(\frac{n}{B}))$ nodes in the proposed linear space interval tree in $O(\log(\frac{n}{B}))$ I/Os such that any interval stabbed by q is associated with one of these nodes.

For query point q and each interval (s_j, e_j) associated with any of the $O(\log(\frac{n}{B}))$ nodes obtained by the above lemma, either $s_j \leq q$ or $q \leq e_j$ is true. The interval stabbing query can now be answered by issuing $O(\log(\frac{n}{B}))$ single-constraint queries (i.e., check if $q \leq e_j$ if $s_j \leq q$ and vice versa) on these nodes. Therefore, Lemma 5.10 can be rewritten as follows.

[7]For any given nodes u_1 and u_2, $stab(u_1) \leq stab(u_2)$ if u_1 comes before u_2 during the in-order traversal of BST.

[8]For any node u, the total number of intervals assigned to nodes in its subtree is $O(size(u))$. This fact follows because (1) all our start and end points are distinct, and (2) for any interval assigned to node u, both its start and end points should be of some value associated with one of its descendants.

Lemma 5.11. A set $\mathcal{I}$ of n intervals can be categorized into subsets using an interval tree structure, such that an interval stabbing query (with two constraints) can be decomposed into $O(\log(\frac{n}{B}))$ queries with a single constraint.

5.5.2 Interval Tree within an Interval Tree

Taking a clue from Lemma 5.11, we aim to decompose top-CRMQ problem into a set of simpler queries. Intuitively, we can maintain an interval tree structure with respect to the backward intervals of all interval-pairs and reduce the original problem (which is a five-constraints query) to $O(\log(\frac{n}{B}))$ four-constraints queries. Each of these four-constraints queries can be further reduced to $O(\log(\frac{n}{B}))$ three-constraints queries by employing another interval tree structure with respect to the forward intervals on a smaller set of interval-pairs. We elaborate on such an interval-tree-within-an-interval-tree approach below to achieve the result summarized in Lemma 5.9.

Data structure: The proposed data structure consists of three components described as follows:
Backward interval tree: This is an interval tree based on backward intervals of all interval-pairs in $\mathcal{I}$ as described earlier in the beginning of this section.
Forward interval trees: The backward interval tree partitions the set $\mathcal{I}$ of interval-pairs into disjoint sets such that each set is associated with some node in the interval tree. Let $\mathcal{I}(u_b)$ be such set associated with node u_b in backward interval tree. We maintain an interval tree at each node u_b based on the forward intervals of all interval-pairs in $\mathcal{I}(u_b)$.
Dominance structures: Let $\mathcal{I}(u_b, v_f)$ be the set of interval-pairs associated with node v_f in forward interval tree that is in turn associated with node u_b in backward interval tree. For each possible set $\mathcal{I}(u_b, v_f)$ we maintain data structures for answering different three-dimensional dominance queries [1] as listed below.

Q_1: (1) $prev(i) < a,$ (4) $b < next(i)$ and (5) $A[i] \geq \tau$

Q_2: (2) $a \leq i,$ (3) $i \leq b$ and (5) $A[i] \geq \tau$

Q_3: (2) $a \leq i,$ (4) $b < next(i)$ and (5) $A[i] \geq \tau$

Q_4: (1) $prev(i) < a,$ (3) $i \leq b$ and (5) $A[i] \geq \tau$

With each of the above components occupying linear space total space required for the proposed data structure can be bounded by $O(|\mathcal{I}|)$ words. Space requirement of the backward interval tree is $O(|\mathcal{I}|)$ words (Lemma 5.10). By the same argument space requirement of a forward interval tree associated with node u_b of backward interval tree is bounded by $O(|\mathcal{I}(u_b)|)$. Thus, the total space required for all forward interval trees is $O(|\mathcal{I}|)$ words. Moreover, since each interval-pair belongs to exactly one of the $\mathcal{I}(u_b, v_f)$ set, all dominance structures collectively occupy linear space.

Query algorithm: We begin by employing the standard interval tree algorithm (Lemma 5.10) to identify $O(\log(|\mathcal{I}|/B))$ nodes in the backward interval tree such that any interval-pair that has its backward interval stabbed by a is associated with one of these $O(\log(|\mathcal{I}|/B))$ nodes. We then apply the same algorithm to each of the forward interval tree associated with these $O(\log(|\mathcal{I}|/B))$ nodes to obtain $O(\log(|\mathcal{I}|/B))$ nodes in a single forward interval tree and $O(\log^2(|\mathcal{I}|/B))$ nodes overall such that any interval-pair that has its backward interval stabbed by a and forward interval stabbed by b is associated with one of these $O(\log^2(|\mathcal{I}|/B))$ nodes. We call these nodes candidate nodes and the set of interval-pairs associated with these nodes candidate sets. We now need to further explore only the retrieved candidate sets to get the desired outputs.

For each candidate node v_f belonging to a forward interval tree that in turn is associated with the node u_b in the backward interval tree, let $stab(v_f)$ and $stab(u_b)$ be the end points maintained at nodes v_f and u_b respectively. Then, each interval-pair in $\mathcal{I}(u_b, v_f)$ is stabbed by $stab(u_b)$ and $stab(v_f)$ on its backward and forward interval respectively. By careful examination of the relative values of $a, b, stab(u_b)$ and $stab(v_f)$, we can eliminate two constraints out of five for top-CRMQ and is one of the crucial observations. We classify node v_f into one the following categories based on which two constraints are satisfied by the interval-pairs in set $\mathcal{I}(u_b, v_f)$:

T_1: $a \leq stab(u_b) \leq stab(v_f) \leq b$

T_2: $stab(u_b) \leq a \leq b \leq stab(v_f)$

T_3: $stab(u_b) \leq a \leq stab(v_f) \leq b$

T_4: $a \leq stab(u_b) \leq b \leq stab(v_f)$

It can be easily verified that each of these categories lead to the query types Q_1, Q_2, Q_3, and Q_4 respectively on set $\mathcal{I}(u_b, v_f)$ to obtain the interval-pairs satisfying all five constraints required for top-CRMQ problem.

Thus, by first obtaining the candidate nodes and then applying appropriate three-dimensional dominance query on each of them all desired outputs can be retrieved. By Lemma 5.10 number of I/Os spent on querying backward interval tree as well as each of the forward interval trees are bounded by $O(\log(|\mathcal{I}|/B))$ I/Os. Therefore, all candidate nodes can be obtained by spending $O(\log^2(|\mathcal{I}|/B))$ I/Os. Moreover, data structure from [1] used for dominance query also requires additional $O(\log_B |\mathcal{I}|)$ I/Os. Hence, total number of I/Os required is $O(\log^2(|\mathcal{I}|/B) \log_B |\mathcal{I}| + \frac{k}{B}) = O(\log^3(|\mathcal{I}|/B) + \frac{k}{B})$. This completes the proof of Lemma 5.9.

5.6 Bootstrapping

The I/O bound in Lemma 5.9 is optimal when $k \geq B \log^3(n/B)$. In the present section, we bootstrap this result to optimally answer "special" top-CRMQ queries. We start by introducing a blocking scheme that forms the basis of all subsequent external memory results.

Blocking scheme: Let blocking factor $\delta_j = B(\log^{(j)}(\frac{n}{B}))^5$ and $k_j = B(\log^{(j)}(\frac{n}{B}))^3$ for $j = 1, 2, 3, ..., \log^*(\frac{n}{B})$. Without loss of generality, we further assume that both δ_j and k_j are always rounded to the next highest power of 2 [3]. We partition the array $A[1...n]$ into $\frac{n}{\delta_j}$ disjoint blocks each of size δ_j such that block $A_{j,t} = A[(t-1)\delta_j + 1...t\delta_j]$. Define $f_{j,t}$ to denote the left boundary of the block $A_{j,t}$. We will say that a block of size δ_j is δ_j-block and a blocking boundary of partitioning based on δ_j (i.e., $f_{j,t}$) is δ_j-boundary. For consistency, fix $\delta_0 = n$ and $A_{0,1} = A[1...n]$. Given a range $[a, b]$, let $A[a^j...b^j]$ be the longest span of δ_j blocks that is completely within $A[a...b]$. Suppose query range $[a, b]$ intersects blocks $A_{j,l}, A_{j,l+1}, ..., A_{j,t}$ then $a^j = f_{j,l+1}$ and $b^j = f_{j,t} - 1$. We prove the results in Lemma 5.12 and Lemma 5.13 in the remainder of this section.

Lemma 5.12. A top-CRMQ (a, b, k, τ) can be answered in $O(\frac{k_{\mu+1}}{B} + \frac{k}{B})$ I/Os using an $O(n \log^* n)$-space structure if the span $A[a...b]$ is completely within a δ_μ-block for $\mu \in [0, \log^*(\frac{n}{B})]$.

[3] In order to ensure δ_{j-1} is always divisible by δ_j.

Proof. For each block $A_{j,t}$, we maintain a data structure $IT_{j,t}$ (of size $|IT_{j,t}|$ words) summarized in Lemma 5.9 [4]. The total space occupancy is $O(\sum_j \sum_t |IT_{j,t}|) = O(n \log^* n)$ space. Then the δ_μ-block containing span $A[a...b]$ i.e., $A_{\mu,t}$ with $t = \lceil \frac{a}{\delta_\mu} \rceil$ can be queried using structure $IT_{\mu,t}$ to obtain the desired answers in $O(\log^3(\frac{\delta_\mu}{B}) + \frac{k}{B}) = O(\frac{k_{\mu+1}}{B} + \frac{k}{B})$ I/Os. □

Lemma 5.13. A top-CRMQ (a, b, k, τ) can be answered in $O(\frac{k_{\mu+1}}{B} + \frac{k}{B} + \log^* n)$ I/Os using an $O(n)$-space structure if the span $A[a...b]$ is completely within a δ_μ-block for $\mu \in [0, \log^*(\frac{n}{B})]$.

The space blowup in Lemma 5.12 comes from the fact that, each interval-pair in $\mathcal{I}$ is repeated $\log^*(\frac{n}{B})$ times as a part $\log^*(\frac{n}{B})$ number of $IT_{\{\cdot,\cdot\}}$'s. We introduce a categorization technique based on the blocking scheme described earlier that avoids this space blowup, though at the cost of (acceptable) slow-down in query performance. We categorize the input interval-pairs in set $\mathcal{I}$ into $\log^*(\frac{n}{B}) + 1$ types based on the following rule: An interval-pair $(i, \cdot, \cdot, \cdot, \cdot)$ is categorized as type-j if its both intervals (i.e., backward and forward) are stabbed by a δ_j-boundary, but at least one of them is not stabbed by a δ_{j-1}-boundary.

Taking into account the boundary conditions, an interval-pair is termed as type-1 if its both intervals are stabbed by a δ_1-boundary, whereas for an interval-pair of type-$(\log^*(\frac{n}{B}) + 1)$, either of its interval is not stabbed by any boundary i.e., i and $prev(i)/next(i)$ are within the same $\delta_{\log^*(\frac{n}{B})}$-block (which is of size $\Theta(B)$). Let n_j represents the number of type-j interval-pairs, then $n_1 + n_2 + ... + n_{\log^*(\frac{n}{B})+1} = n$.

We now describe the data structure and query algorithm to achieve the result in Lemma 5.13. Intuitively, our idea is to make separate linear space data structures for interval-pairs in each type thereby restricting the total space to $O(n)$ words. However, now we need to probe multiple structures to retrieve all answers, thus, incurring an additive $\log^*(\frac{n}{B})$ term in the query I/Os.

Data structure: We maintain the following structures. As each of the components described below occupies $O(n)$ words the overall space requirement is linear.

[4] $IT_{j,t}$ is the structure in Lemma 5.9 over the following set of interval-pairs $\mathcal{I}_{j,t} = \{(i, \cdot, \cdot, \cdot, \cdot) \in \mathcal{I} | i \in [(t-1)\delta_j + 1, t\delta_j]\}$.

- For each block $A_{j,t}$ maintain a structure $IT_{j,t}$ summarized in Lemma 5.9 by considering only type-$(j+1)$ and type-$(j+1)$ interval-pairs occupying $O(\sum_j (n_{j+1} + n_{j+2})) = O(n)$ space.

- We create a collection of two-dimensional points by mapping each type-j interval-pair $(i, A[i], prev(i), next(i))$ to a point $(i, A[i])$. Then we apply rank-space reduction to these two-dimensional points and maintain a three-sided range reporting structure TS_j by Larsen et al. [85] on this collection. All those type-j interval pairs with $i \in [a, b]$ and $A[i] \geq \tau$ for any given a, b, and τ can be answered in optimal I/Os using TS_j. Further, we associate each two-dimensional point with its corresponding interval-pair, so that the interval-pairs corresponding to the points reported by structure from [85] can be obtained without spending any additional I/Os. Moreover, to be able to query data structure in [85] we need to map the boundary points (a and b) and the threshold τ to rank-space. This can be achieved in constant time by maintaining two bit vectors (along with rank-select structure [116]) of length n. Total space required for this component is bounded by $O(n_j)$ words + $O(n)$ bits = $O(n_j + \frac{n}{\log n})$ words. Thus, overall space corresponding to $j = 0, 1, 2, ..., \log^*(\frac{n}{B}) + 1$ is $O(n)$ words.

- We also maintain a list A' of all interval pairs $(i, \cdot, \cdot, \cdot, \cdot)$ in the ascending order of i. Space occupancy is $O(n)$ words.

Query algorithm: As before, let $A_{\mu,t}$ with $t = \lceil \frac{a}{\delta_\mu} \rceil$ be the δ_μ-block containing $A[a...b]$. Then we query $IT_{\mu,t}$ by spending $O(\frac{k_{\mu+1}}{B} + \frac{k}{B})$ I/Os. However, this will give only the outputs of type $(\mu+1)$ and $(\mu+2)$. It remains to show how to retrieve the outputs of type-h, for $h \leq \mu$ or $h \geq \mu + 3$.

We first demonstrate how type-h outputs with $h \leq \mu$ are retrieved when span $A[a...b]$ is known to be completely within the a δ_μ block i.e., $A_{\mu,t}$. We note that for any type-h link $(i, \cdot, \cdot, \cdot, \cdot)$ with $h \leq \mu$ and i falling within the block $A_{\mu,t}$ (i.e., $i \in [f_{\mu,t}, f_{\mu,t+1} - 1]$), both its forward as well as backward intervals are stabbed by δ_μ-boundaries ($f_{\mu,t}$ and $f_{\mu,t+1}$ respectively). Therefore, such an interval-pair implicitly satisfies constraints $prev(i) < a$, $b < next(i)$. Hence, for $h \leq \mu$ we only need to take into account the position and weight constrain of the interval-pair (i.e., $i \in [a, b]$ and $A[i] \geq \tau$) and all such type-h outputs can be obtained in optimal I/Os by querying structure

TS_h. Therefore, overall I/Os required for retrieving all type-h outputs for $h \leq \mu$ are bounded by $O(\mu + \frac{k}{B}) = O(\log^*(\frac{n}{B}) + \frac{k}{B})$.

Finally all type-h outputs for $h \geq \mu + 3$ can be efficiently retrieved using the following key observation. Any type-h interval-pair $(i, \cdot, \cdot, \cdot, \cdot)$, with $h \geq \mu + 3$ is an output, only if i falls within a $\delta_{\mu+1}$-block that contains either a or b, otherwise at least one of two conditions $prev(i) < a$, $b < next(i)$ will be violated. Therefore, the number of candidate interval-pairs in this case is only $2\delta_{\mu+2}$, and the output interval-pairs can be obtained by scanning the two $\delta_{\mu+2}$-blocks in A' to evaluate the five conditions listed in Observation 5.8 for each of the candidate. The I/Os required in this step are bounded by $O(\frac{\delta_{\mu+2}}{B}) = o(\frac{k_{\mu+1}}{B})$.

Putting together all pieces, the number of I/Os required to answer a top-CRMQ (a, b, k, τ) with $A[a...b]$ completely within a δ_μ-block, can be bounded by $O(\frac{k_{\mu+1}}{B} + \frac{k}{B} + \log^* n)$.

5.7 The Final Data Structures

This section is dedicated for proving Theorem 5.2 and Theorem 5.3. Given a top-CRMQ (a, b, k), the structure presented in Lemma 5.9 can be maintained in $O(n)$-space to optimally handle queries with $k = \Omega(B \log^3(n/B))$. Otherwise, we find the parameter $\pi \in [1, \log^*(n/B)]$, where $k_{\pi+1} < k \leq k_\pi$ (assume $k_{\log^*(n/B)+1} = 0$). Then we decompose the original query into following subqueries:

- top-CRMQ $(a, a^\pi - 1, k, \tau)$
- top-CRMQ (a^π, b^π, k)
- top-CRMQ $(b^\pi + 1, b, k, \tau)$

Here $A[a^\pi...b^\pi]$ represents the longest span of δ_π blocks that is completely within $A[a...b]$. Let Out_i represents the set of answers corresponding to the above queries for $i = 1, 2, 3$ (procedure to obtain them will be described later). Notice that these are disjoint sets and cardinality of each of them is $O(k)$. Moreover, $\cup_{i=1}^3 Out_i$ is a superset of final answers for the original query (a, b, τ). Therefore, those interval-pairs $(i, A[i], C[i], prev(i), next(i)) \in \cup_{i=1}^3 Out_i$ with $prev(i) < a$, $next(i) > b$ and $A[i] \geq \tau$ can be ***uniquely*** reported as the final answers (condition $i \in [a, b]$ is satisfied implicitly).

It remains to show how to retrieve the output set for each of the subqueries efficiently. Both Out_1 and Out_3 can be obtained in $O(k_{\pi+1}/B + k/B) = O(1 + k/B)$ I/Os by maintaining an $O(n \log^* n)$-space structure (refer to Lemma 5.12). By querying on the structure described in the following lemma, Out_2 also can be obtained in optimal I/Os. This completes the proof of Theorem 5.2.

Lemma 5.14. By maintaining an $O(n \log^* n)$-space structure, a top-CRMQ (α, β, K) can be answered in optimal $O(1 + K/B)$ I/Os if $A[\alpha...\beta]$ is a span of several δ_π-blocks and $K \leq k_\pi$ for $\pi \in [0, \log^*(\frac{n}{B})]$.

Similarly, using the linear space structure in Lemma 5.13, both Out_1 and Out_3 can be obtained in $O(k_{\pi+1}/B + k/B + \log^* n) = O(\log^* n + k/B)$ I/Os. Combining this with the following lemma for retrieving Out_2, we achieve the result summarized in Theorem 5.3.

Lemma 5.15. By maintaining an $O(n)$-space structure, a top-CRMQ (α, β, K) can be answered in $O(\log^* n + K/B)$ I/Os if $A[\alpha...\beta]$ is a span of several δ_π-blocks and $K \leq k_\pi$ for $\pi \in [0, \log^*(\frac{n}{B})]$.

The remaining part of this section is dedicated to prove these two lemmas i.e., Lemma 5.14 and 5.15. We identify the parameter θ as the smallest i such that, there exists a δ_i-boundary in $[\alpha, \beta]$. Using θ we decompose top-CRMQ (α, β, K) further into the following subqueries, and obtain the desired answers by merging the outputs of individual subqueries. Here $A[\alpha^\theta...\beta^\theta]$ represents the longest span of δ_θ blocks that is completely within $A[\alpha...\beta]$.

- Q_{left}: top-CRMQ $(\alpha, \alpha^\theta - 1, K)$
- O_{middle}: top-CRMQ $(\alpha^\theta, \beta^\theta, K)$
- Q_{right}: top-CRMQ $(\beta^\theta + 1, \beta, K)$

5.7.1 Answering Q_{middle}

Starting from left boundary of each block $A_{j,t}$ i.e., $f_{j,t}$, consider the spans covering $1, 2, 4, 8, ...$ blocks of size δ_j such that it does not cross the first δ_{j-1}-boundary that follows $f_{j,t}$. We maintain the top-k_j answers (i.e., the corresponding pentuples) for each of these spans explicitly (in descending order of weight) i.e., we maintain the list $ML(j, t, i)$ that contains the answers for top-CRMQ

with k_j as an input on the span $A[f_{j,t}...f_{j,t+2^i}-1]$ for any $1 \leq j \leq \log^*(\frac{n}{B})$, $1 \leq t \leq \frac{n}{\delta_j}$ and $i = 0, 1, 2, ..., \log(\frac{\delta_{j-1}}{\delta_j})$. Overall space requirement for such a storage is $O(\sum_j (\frac{n}{\delta_j}) k_j \log(\frac{\delta_{j-1}}{\delta_j}))$ $= O(\sum_j \frac{n}{\log^{(j)}(\frac{n}{B})}) = O(n)$ words.

To answer Q_{middle}, we represent $A[\alpha^\theta...\beta^\theta]$ as union of two overlapping spans each of which covers 2^i δ_θ-blocks for some integer i. Let $[f_{\theta,l'}, f_{\theta,l'+2^i}-1]$ and $[f_{\theta,t'-2^i}, f_{\theta,t'}-1]$ be the ranges for these overlapping spans such that $f_{\theta,l'} = \alpha^\theta$ and $f_{\theta,t'} - 1 = \beta^\theta$. It is evident that any top-K answer for $A[\alpha^\theta...\beta^\theta]$ should also be in top-K answers of either of the overlapping spans i.e., it should be present in either $ML(j, l', i)$ or $ML(j, t'-2^i, i)$. Top-K answers (in sorted order) for these two overlapping spans can be directly retrieved from the maintained precomputed answers in $O(\frac{k}{B})$ I/Os. Further, the two lists can be merged to obtain the outputs for Q_{middle} by a simple scan. However, before merging we discard any answer belonging to the region of overlap between two ranges (i.e., span $A[f_{\theta,t'-2^i}...f_{\theta,l'+2^i}-1]$) from either of the answer lists to ensure uniqueness of the reported answers. In conclusion, Q_{middle} can be answered optimally using an $O(n)$-space structure.

5.7.2 Answering Q_{left} and Q_{right}

I/O-optimal structure: For each $A_{j,t}$ and $h < j$ we maintain top-k_j answers (in descending order of weight) for the span bounded by $f_{j,t}$ and the first δ_h-boundary that follows $f_{j,t}$. Similarly, top-k_j answers for the span bounded by $f_{j,t+1}-1$ and the first δ_h-boundary that precedes it are maintained. These answers are maintained in two lists SL_r and SL_l. The list $SL_r(j,t,h)$ and $SL_l(j,t,h)$ contains the answer to top-CRMQ with k_j as an input on the span $[f_{j,t}, f_{h,t'+1}-1]$ and $[f_{h,t'}, f_{j,t+1}-1]$ respectively for any $1 \leq j \leq \log^*(\frac{n}{B})$, $1 \leq t \leq \frac{n}{\delta_j}$ and $h < j$ with $t' = \lceil \frac{t}{(\delta_h/\delta_j)} \rceil$. Here t' is the δ_h-block that contains the δ_j-block t. Overall space for maintaining these inter-level answers can be bounded by $O(\sum_j \frac{n}{\delta_j} k_j(j-1)) = O(\sum_j \frac{nj}{(\log^{(j)}(\frac{n}{B}))^2}) = O(n \log^* n)$ words. Desired answers for the top-CRMQ query on spans $A[\alpha...\alpha^\theta - 1]$ and $A[\beta^\theta + 1...\beta]$ are simply the first K entries in the appropriate lists $SL_r(\pi, \cdot, \theta)$, $SL_l(\pi, \cdot, \theta)$ respectively and the I/Os needed for retrieving are $O(\frac{K}{B})$. Combing this result along with $O(n)$-space structure capable of answering Q_{middle}, we prove Lemma 5.14.

Linear space structure: To achieve linear space, we do the following modification to the data structure just described: maintain $SL_r(j,\cdot,\cdot)$ and $SL_l(j,\cdot,\cdot)$ only for those $j \leq \phi \leq \log^*(\frac{n}{B})$, where $\log^{(\phi)}(\frac{n}{B}) \geq \log^*(\frac{n}{B}) > \log^{(\phi+1)}(\frac{n}{B})$. Then space can be bounded by $O(\frac{n}{(\log^{(2)}(\frac{n}{B}))^2} + \frac{2n}{(\log^{(3)}(\frac{n}{B}))^2} + \frac{3n}{(\log^{(4)}(\frac{n}{B}))^2} + ... + \frac{(\phi-1)n}{(\log^{(\phi)}(\frac{n}{B}))^2}) = O(\frac{n}{\log^*(\frac{n}{B})})$ words. In addition, we maintain all $SL_r(\phi+1,\cdot,\phi)$ and $SL_l(\phi+1,\cdot,\phi)$ as well occupying $O(\frac{n}{(\log^{(\phi+1)}(\frac{n}{B}))^2}) = o(n)$ words. Further, we also assume the availability of the linear space data structure described in Lemma 5.13. Thus overall space is bounded by $O(n)$-words. In order to answer a query, we consider the following cases:

1. *If* $\pi \leq \phi$: Obtain answers from the appropriate $SL_r(\pi,\cdot,\theta)$ and $SL_l(\pi,\cdot,\theta)$ in $O(\frac{K}{B})$ I/Os.

2. *If* $\pi = \phi + 1$: Obtain answers from appropriately chosen lists $SL_r(\phi+1,\cdot,\phi)$, $SL_r(\phi,\cdot,\theta)$ and then merge them by spending $O(\frac{K}{B})$ I/Os. Similarly appropriate lists $SL_l(\phi+1,\cdot,\phi)$, $SL_l(\phi,\cdot,\theta)$ can be accessed to obtain the desired results.

3. *If* $\pi > \phi + 1$: We first obtain answers for the span $A[\alpha^{\phi+1}...\alpha^{\theta}-1]$ and $A[\beta^{\theta}+1...\beta^{\phi+1}]$ from appropriate SL_r and SL_l structures in $O(\frac{K}{B})$ I/Os. Whereas answers for $A[\alpha...\alpha^{\phi+1}-1]$ (resp., $A[\beta^{\phi+1}+1...\beta]$) can be obtained in $O(\log^3(\frac{\delta_{\phi+1}}{B}) + \frac{K}{B} + \log^*(\frac{n}{B})) = O(\log^*(\frac{n}{B}) + \frac{K}{B})$ I/Os as it is completely within a block of size $\delta_{\phi+1}$ (from Lemma 5.13).

Therefore, total number of I/Os required to answer Q_{left} and Q_{right} are bounded by $O(\log^*(\frac{n}{B}) + \frac{K}{B})$, when linear space data structure is used. Result summarized in Lemma 5.15 can now be obtained by using this structure in addition to $O(n)$-space structure for answering Q_{middle}.

5.8 CRMQ in Internal Memory

In this section, we show how to modify our external memory data structures to achieve the result in Theorem 1. We first obtain internal memory version of Lemma 5.9 by simply subsisting B by 2. Recall that this result is obtained by querying $O(\log^2 n)$ three-dimensional dominance structures. By using our new dominance structure (Theorem 5.4) instead of the one by Afshani [1], the outputs from each of those three-dimensional dominance queries can be obtained in the sorted order. Moreover, these outputs can be merged to get a complete list of all answers in sorted order using a heap

structure. For our purpose, we use an atomic heap [48] that can perform all heap operations in $O(1)$ in RAM model provided the heap size is $\log^{O(1)} n$. By putting everything together, we obtain an $O(n)$-word space and $O(\log^3 n + k)$ query time data structure for the sorted version of Problem 2.

We now apply blocking scheme with a single blocking factor $\delta_1 = \log^4 n$, and maintain the interval-tree based structure over each block $A_{1,t} = A[(t-1)\delta_1 + 1...t\delta_1]$ as $IT_{1,t}$, taking overall $O(n)$ space. Recall that $\delta_0 = n$ and we also maintain $IT_{0,1}$. Further we maintain, the structures $ML(\cdot,\cdot,\cdot)$ as described in Section 5.7.1 occupying $O(n)$ word space i.e., from each δ_1-boundary $f_{1,t}$ consider the spans covering $1, 2, 4, 8, ...$ δ_1-blocks and maintain top-k_1 answers ($k_1 = \log^3 n$) for each of these spans explicitly. Whenever query input $k \geq \log^3 n$, it can be answered optimally using $IT_{0,1}$. For $k < \log^3 n$ and the input range $[a, b]$ completely within a δ_1-block, query can be answered in $O(\log^3 \log n + k)$ time only using appropriate $IT_{1,t}$ structure. Otherwise, we can retrieve top-k answers from fringe spans $A[a...a^1 - 1]$, $A[b^1...b]$ and a middle span $A[a^1...b^1 - 1]$ (refer Section 5.7.1, 5.7.2) and merge them to report final top-k answers with identical query time of $O(\log^3 \log n + k)$. The non-optimal $O(\log^3 \log n)$-additive factor is due to the time for querying the interval tree based structure maintained over each δ_1 block. Therefore, for improving the case where $k < \log^3 \log n$ and the query span $A[a...b]$ is completely within a δ_1 blocks, we maintain the following additional structure. Given a δ_1-block $A_{1,t}$, for every span $A[f_{1,t} + i, f_{1,t} + i + 2^j - 1]$ for $i = 0, 1, 2, 3, ..., (\delta_1 - 1)$ and $j = 0, 1, 2, ..., \log \delta_1$, maintain top-$(\log^3 \log n)$ answers (in sorted order). Instead of explicitly maintaining, an output element $A[r]$ (or its location r) for a particular span, we simply encode it as an offset from the left boundary of the span i.e., $r - f_{1,t} + i$ in $O(\log \delta_1) = O(\log \log n)$ bits. Thus, overall space requirement can be bounded by $O(n \log^2 \delta_1) = o(n)$ bits. Now any span $A[a...b]$ with both a as well as b in the same δ_1-block can be partitioned into two overlapping spans $A[a...y]$ and $A[x...b]$ where $a < x \leq y < b$, such that the top-k answers of these overlapping spans are precomputed and can be retrieved in optimal time. Finally, by merging these answers, we obtain the final output.

Sorted three-dimensional dominance reporting: Our data structure for proving Theorem 5.4 is based on the same approach as in [29, 92]. But we will also need additional ideas to output points in

sorted order. We associate sets of points $P(v)$ with nodes v of a binary tree T. Let $\max_{xy}(S)$ denote those points of a set S whose projections on the xy-plane are maximal. We set $S(w_r) = S$ for the root w_r of T. In every node v starting with the root, we store set $P(v) = \max_{xy} S(v)$. Then, we divide all points from $S(v) \setminus P(v)$ into two equal parts according to their z-coordinates and associate them with children v_l, v_r of v. In other words, points from $S(v) \setminus P(v)$ are distributed among $S(v_l)$ and $S(v_r)$ so that (1) $p_l.z < p_r.z$ for any $p_l \in S(v_l)$ and $p_r \in S(v_r)$, (2) $|S(v_r)| \leq |S(v_l)| \leq |S(v_r)| + 1$. Finally, we recursively apply the same procedure to $S(v_l)$ and $S(v_r)$.

For every node v, we keep all points of $P(v)$ sorted by their x-coordinates in an array $A(v)$. We maintain a data structure from [19] that supports sorted reporting queries on $A(v)$: for any query interval $[a, b]$, $D(v)$ reports all points $p \in A[i]$, such that $a \leq i \leq b$ and $p.z \geq c$, sorted in decreasing order of their z-coordinates. $D(v)$ uses $O(|P(v)|)$ space and answers queries in $O(k+1)$ time, where k is the number of reported points. We also store structures $D_x(v)$, $D_y(v)$ so as to enable us to answer predecessor and successor queries on x, y-coordinates of points in $P(v)$.

Using $D(v)$, $D_x(v)$, and $D_y(v)$, we can answer a sorted dominance query $Q = [a, +\infty] \times [b, +\infty] \times [c, +\infty]$ on $P(v)$. Since $P(v)$ contains maximal points with respect to their x- and y-coordinates, all p_1, $p_2 \in P(v)$ have the following property: if $p_1.x > p_2.x$, then $p_1.y < p_2.y$, i.e., y-coordinates of points in $P(v)$ decrease monotonously with increasing x-coordinates. Let p_l be the point in $P(v)$ with the smallest x-coordinate, such that $p_l.x \geq a$; let p_r be the point in $P((v)$ with the smallest y-coordinate, such that $p_r.y \geq b$. Let i_l and i_r denote the x-ranks[9] of p_l and p_r respectively. All points p stored in $A[i_l...i_r]$ and only those points satisfy $p.x \geq a$ and $p.y \geq b$. Hence, we can answer a query Q on $P(v)$ by reporting all points in $A[i_l...i_r]$ in decreasing order of their z-coordinates until all points p, $p.z \geq c$, are output.

The same sorted dominance query on S is answered as follows. Let Π_q denote the search path for c in T. We report all points $p \in P(v)$ for all nodes $v \in \Pi_q$. For every node u that is a right sibling of $v \in \Pi_q$, we must report relevant points stored in u and its descendants. First, we answer the dominance queries on $P(u)$; if at least one point was reported, we visit both children of u and

[9]The x-rank of a point p in a set P is the number of points $p' \in P$ such that $p'.x \leq p.x$.

recursively process them. Let $L(u)$ denote the list of points in $P(u) \cap Q$ sorted by their z-coordinates. The union of $L(u)$ for all visited nodes u contains all points in $S \cap Q$: all points p, $p.z \geq c$, are stored in nodes $v \in \Pi_q$ or in right siblings of nodes $v \in \Pi_q$ and their descendants. Our procedure visits all nodes $v \in \Pi_q$ and their right siblings; our procedure also visits all descendants of the right siblings that contain at least one point $p \in Q$, as can be concluded from the following observation.

Observation 1. Suppose that u is the right sibling of some node $v \in \Pi_q$ or a descendant of the right sibling of some $v \in \Pi_q$. If $P(u) \cap Q = \emptyset$, then $P(w) \cap Q = \emptyset$ for all descendants w of u.

Every list $L(u)$ is generated in $O(|L(u)| + 1)$ time: using fractional cascading, we can find indices i_l and i_r in any visited node u in constant time. When i_l and i_r are known, data structure $D(u)$ reports all points $p \in A(u)$, $p.z \geq c$ in $O(|L(u)| + 1)$ time. The total number of nodes u for which lists $L(u)$ were generated is bounded by $O(\log n + k)$. Hence, the total time needed to generate all lists $L(u)$ is $O(\log n + k)$.

It remains to show how to merge all $L(u)$ so that the output is sorted by z-coordinates. We will say that a node u is situated to the right of a node v if u and v are stored in respectively the right and the left subtrees of their lowest common ancestor.

Observation 2. If $p_u.z > p_w.z$ for some $p_u \in P(u)$ and $p_w \in P(w)$, then u is an ancestor of w or u is situated to the right of w in T.

Let V denote the set of all visited nodes. Since the height of T is $O(\log n)$, we can use sweepline approach for sorting points in the query range: we maintain the *current path* Π_c, and report points stored in $P(u)$, $u \in \Pi_c$, in sorted order. Suppose that we work with the current path Π_c at some time. Then this means that all nodes $u \in V$ to the right of Π_c were already processed and points from lists $L(u)$ are already in sorted order.

To initialize the path Π_c, we start at the root and move down the tree until a leaf is reached or the currently visited node u has no child $u_i \in V$. In every visited node u, we move to its right child u_r if $u_r \in V$; otherwise, we move to its left child u_l. Thus Π_c is initialized to the rightmost path that consists of nodes $u \in V$.

We extract the first point (i.e., the point with the highest z-coordinate) from every $L(u)$, $u \in \Pi_c$, and insert them into a priority queue Q. The following steps are repeated until all points in all $L(u)$, $u \in V$, are sorted. We extract the highest point p from Q and add it to the sorted list of points. If the list $L(u)$, such that $p \in L(u)$, is not empty, we extract the next point p' from $L(u)$ and add it to Q. When some list $L(w)$, $w \in \Pi_c$, becomes empty, we might need to update the path Π_c. If $L(w)$ is empty and w is the lowest node in Π_c, we remove w from Π_c. If w is the right child of its parent and its left sibling v is in V, we also append new nodes to Π_c. This is done by traversing a downward path that starts in v. In every visited node u, starting with v, we add u to Π_c and move down the tree if at least one child of u is in V; if both children of u are in V, we always select the right child. For every new node u in Π_c, we extract the highest point $p \in L(u)$ and add it to Q. Otherwise, if w has no left sibling or the left sibling of w is not in Π_c, then we move up in the tree and consecutively examine all ancestors w' of w starting with the parent. If $L(w')$ for an ancestor w' of w is empty, we remove w' from Π_c. If w' has a left sibling $w'' \in V$, we append the rightmost path starting at w'' to Π_c as described above. Otherwise, we examine the ancestors of w' until a node u, $L(u) \neq \emptyset$, is reached. When Π_c and Q are empty, we have generated the sorted list of all points in $S \cap Q$. Correctness of our procedure follows from Observation 2. Suppose that a point $p_1 \in L(u_1)$ was reported before $p_2 \in L(u_2)$, then either (1) u_1 is to the right of u_2, or (2) u_1 is an ancestor of u_2, or (3) u_2 is ancestor of u_1. In the case (1) $p_1.z \leq p_2.z$ by Observation 2. In the case (2) u_1 is an ancestor of u_2. If p_1 was reported before u_2 was inserted into Π_c, then $p_1.z \geq p_3.z$ for some $p_3 \in L(u_3)$, where u_3 is to the right of u_2. Hence, $p_1.z \geq p_3.z \geq p_2.z$. If p_1 was reported after u_2 had been included into Π_c, then it follows from the description that $p_1.z \geq p_2.z$. Case (3) is identical with the second part of case (2).

We implement Q using the atomic heap data structure [48]; Since Q contains $O(\log n)$ elements, all operations on Q can be supported in $O(1)$ time. By keeping the depths of all non-empty nodes $u \in \Pi_c$ in another atomic heap, we can determine whether there are non-empty nodes $u' \in \Pi_c$ below a given node u in $O(1)$ time. Thus, we can sort all points $p \in L(v)$, $v \in V$, by their z-coordinates in $O(|V| + \sum_{v \in V} |L(v)|)$ time. This completes the proof of Theorem 5.4.

5.9 Summary

In this chapter we introduced the problem of colored (categorical) range maxima that generalizes the fundamental problem of computing maxima in a query range to the colored scenario. We provide an optimal solution of the colored range maxima problem in internal memory. Our external memory data structure uses $O(n)$ space and answers queries in $O(\log^* n + k/B)$ I/Os. We show that this problem is generalizes the problem of three-sided categorical range reporting. The proposed data structure enables us to enumerate all outputs of a three-sided categorical range reporting uniquely, thus, closing one of the important open problems in this research area.

Chapter 6
Ranked Retrieval in Uncertain Databases

6.1 Introduction

The efficient processing of uncertain data is an important issue in many application domains because of the imprecise nature of data they generate. The nature of uncertainty in data is quite varied, and often depends on the application domain. In response to this need, much efforts have been devoted to modeling uncertain data [133, 35, 31, 83, 121]. Most models have been adopted to possible world semantics, where an uncertain relation is viewed as a set of possible instances (worlds) and correlation among the tuples governs generation of these worlds.

Consider traffic monitoring application data [125] (with modified probabilities) as shown in Table 6.1, where radar is used to detect car speeds. In this application, data is inherently uncertain because of errors in reading introduced by nearby high voltage lines, interference from near by car, human operator error etc. If two radars at different locations detect the presence of the same car within a short time interval, such as tuples t_2 and t_4 as well as t_3 and t_6, then at most one radar reading can be correct. We use x-relation model to capture such correlations. An x-tuple τ specifies a set of exclusive tuples, subject to the constraint $\sum_{t_i \in \tau} Pr(t_i) \leq 1$. The fact that t_2 and t_4 cannot be true at the same time, is captured by the x-tuple $\tau_1 = \{t_2, t_4\}$ and similarly $\tau_2 = \{t_3, t_6\}$. Probability of a possible world is computed based on the existence probabilities of tuples present in

TABLE 6.1. Traffic monitoring data

t_1 ,$\{t_2, t_4\}$, $\{t_3, t_6\}$, t_5

Time	Car Location	Plate Number	Speed	Probability	Tuple Id
11:55	L1	Y-245	130	0.30	t_1
11:40	L2	X-123	120	0.40	t_2
12:05	L3	Z-541	110	0.20	t_3
11:50	L4	X-123	105	0.50	t_4
12:10	L5	L-110	95	0.30	t_5
12:15	L6	Z-541	80	0.45	t_6

a world and absence probabilities of tuples in the database that are not part of a possible world. For example, consider the possible world $pw = \{t_1, t_2, t_3\}$. Its probability is computed by assuming the existence of t_1, t_2, t_3, and the absence of t_4, t_5, and t_6. However, since t_2 and t_4 are mutually exclusive, presence of tuple t_2 implies absence of t_4 and same is applicable for tuples t_3 and t_6. Therefore, $Pr(pw) = 0.3 \times 0.4 \times 0.2 \times (1 - 0.3) = 0.0168$.

Top-k queries on a traditional certain database have been well studied. For such cases, each tuple is associated with a single score value assigned to it by a scoring function. There is a clear total ordering among tuples based on score, from which the top-k tuples can be retrieved. However, for answering a top-k query on uncertain data, we have to take into account both, ordering based on scores and ordering based on existence probabilities of tuples. Depending on how these two orderings are combined, various top-k definitions with different semantics have been proposed in recent times. Most of the existing work is focused only on the problem of answering a top-k query on a static uncertain data. Though the query time of an algorithm depends on the choice of a top-k definition, linear scan of tuples achieves the best bound so far. Therefore, recomputing top-k answers in an application with frequent insertions and deletions can be extremely inefficient. In this chapter, we present a truly dynamic structure of size $O(n)$ that always maintains the correct answer to the top-k query for an uncertain database of n tuples. The structure is based on a decomposition of the problem so that updates can be handled efficiently. Our structure can answer the top-k query in $O(k \log n)$ time, handle update in $O(\log n)$ time.

6.2 Top-k Queries on Uncertain Data

Soliman et al. [125] first considered the problem of ranking tuples when there is a score and probability for each tuple. Several other definitions of ranking have been proposed since then for probabilistic data.

- Uncertain top-k (U-Topk) [125]: It returns a k-tuple set that appears as top-k answer in possible worlds with maximum probability.
- Uncertain Rank-k (U-kRanks) [125]: It returns a tuple for each i, such that it has maximum probability of appearing at rank i across all possible worlds.

- Probabilistic Threshold Query (PT-k) [67]: It returns all the tuples with probability of appearing in top-k greater than a user specified threshold.
- Expected Rank (E-Rank) [32]: k tuples with the highest value of expected rank $er(t_i) = \sum Pr(pw) rank_{pw}(t_i)$ are returned, where $rank_{pw}(t_i)$ denotes rank of t_i in a possible world pw. In case t_i does not appear in possible world, $rank_{pw}(t_i)$ is defined as $|pw|$.
- Quantile Rank (Q-Rank) [76]: k tuples with lowest value of quantile rank $(qr_\phi(t_i))$ are returned. The ϕ-quantile rank of t_i is the value in the cumulative distributive function (cdf) of rank(t_i), denoted as $cdf(rank(t_i))$ that has a cumulative probability of ϕ. Median rank is a special case of ϕ-quantile rank where $\phi = 0.5$.
- Expected Score (E-Score) [32]: k tuples with the highest value of expected score $es(t_i) = Pr(t_i) score(t_i)$ are returned.
- Parameterized Ranking Function (PRF) [90]: PRF in its most general form is defined as,

$$\Upsilon(t_i) = \sum_r w(t_i, r) \times Pr(t_i, r) \tag{6.1}$$

where w is the weight function that maps a given tuple-rank pair to a complex number and $Pr(t_i, r)$ denotes the probability of a tuple t_i being ranked at position r across all possible worlds. A top-k query returns those k tuples with the highest Υ values. Different weight functions can be plugged in to the above definition to get a range of ranking functions, subsuming most of top-k definitions listed above. A special ranking function $PRF^e(\alpha)$ is obtained by choosing $w(t_i, r) = \alpha^{r-1}$, where α is a constant. Experimental study in [90] reveals that for some value of α with the constraint $\alpha < 1$, PRF^e can approximate many existing top-k definitions. These experiments use Kendall distance [40] between two top-k answers as a measure to compare the ranking functions. The "uni-valley" nature of the graphs obtained by plotting Kendall distance versus varying values of α for various ranking functions in [90] suggests there exists a value of α for which the distance of a particular ranking function to PRF^e is very small i.e., $PRF^e(\alpha)$ can approximate that function quite well.

Algorithms for computing top-k answers using the above ranking functions have been studied for static data. Any changes in the underlying data forces re-computation of query answers. To understand the impact of a change on top-k answers, we analyze relative ordering of the tuples before and after a change, based on these ranking functions.

Let $T = t_1, t_2, .., t_n$ denote independent tuples sorted in non-increasing order of their score. We choose insertion of a tuple as a representative case for changes in T, and monitor its impact on relative ordering of a pair of tuples (t_i, t_j). For ranking function `U-kRanks` ordering of tuples (t_i, t_j) may or may not be preserved by insertion and cannot be guaranteed when the score of a new tuple is higher than that of t_i and t_j. Consider a database $T = t_1, t_2, t_3$ with existence probability values 0.1, 0.5, and 0.2 respectively. When all tuples are independent, probability that tuple t_i appears at rank 2 across all possible worlds is given by $Pr(t_i, 2) = p_i \sum_{x=1}^{i-1}(p_x \prod_{y=1,y\neq x}^{i-1}(1 - p_y))$ [125]. Hence $Pr(t_2, 2) = 0.05 < Pr(t_3, 2) = 0.1$ and tuple t_3 would be returned as an answer for U-2Ranks query. Insertion of a new tuple t_0 with existence probability 0.25 and score higher than that of t_1, causes relative ordering of tuples t_2, t_3 to be reversed as after insertion $Pr(t_2, 2) = 0.15 > Pr(t_3, 2) = 0.0975$. Thus, existing top-$k$ answers do not provide any useful information for re-computation of query answers making it necessary to go through all the tuples again for re-computation in the worst case. Ranking functions `PT-k, E-Rank, Q-Rank` may also result in such relative ordering reversal. However, when tuples are ranked using $PRF^e(\alpha)$, the scope of disturbance in the relative ordering of tuples is limited as explained in later sections. This enables efficient handling of updates in the database. Therefore, this ranking function is well suited for answering top-k queries on a dynamic collection of tuples.

6.3 Problem Statement

Given an uncertain relation T such that each tuple $t_i \in T$ is associated with a membership probability value $Pr(t_i) > 0$ and a score $score(t_i)$, the goal is to retrieve the top-k tuples. Without loss of generality, we assume all scores to be unique and let $t_1, t_2, ..., t_n$ denotes ordering of the tuples in T when sorted in descending order of the score ($score(t_i) > score(t_{i+1})$). We use the parameterized ranking function $PRF^e(\alpha)$ proposed by [90] in this chapter. $PRF^e(\alpha)$ is defined as,

$$\Upsilon(t_i) = \sum_r \alpha^{r-1} \times Pr(t_i, r) \tag{6.2}$$

where α is a constant and $Pr(t_i, r)$ denotes the probability of a tuple t_i being ranked at position r across all possible worlds[1]. A top-k query returns the k tuples with highest Υ values. We refer to $\Upsilon(t_i)$ as the `rank-score` of tuple t_i. In this work, we adopt the x-relation model to capture correlations. An x-tuple τ specifies a set of exclusive tuples, subject to the constraint $Pr(\tau) = \sum_{t_i \in \tau} Pr(t_i) \leq 1$. In a randomly instantiated world τ takes t_i with probability $Pr(t_i)$, for $i = 1, 2, ..., |\tau|$ or does not appear at all with probability $1 - \sum_{t_i \in \tau} Pr(t_i)$. Here $|\tau|$ represents the number of tuples belonging to set τ. Let $\tau(t_i)$ represents an x-tuple to which tuple t_i belongs to. In x-relation model, T can be thought of as a collection of pairwise-disjoint x-tuples. As there are total n tuples in an uncertain relation T, $\sum_{\tau \in T} |\tau| = n$. From now onwards we represent $Pr(t_i)$ by short notation p_i for simplicity.

6.4 Computing $PRF^e(\alpha)$

In this section, we derive a closed form expression for the `rank-score` $\Upsilon(t_i)$, followed by an algorithm for retrieving the top-1 tuple from a collection of tuples. In the next section we show that this approach can be easily extended to a data structure for efficiently retrieving top-k tuples from a dynamic collection of tuples. We begin by assuming tuple independence and then consider correlated tuples, where correlations are represented using x-tuples.

6.4.1 Assuming Tuple Independence

When all tuples are independent, tuple t_i appears at position r in a possible world pw if and only if exactly $(r-1)$ tuples with a higher score value appear in pw. Let $S_{i,r}$ be the probability that a randomly generated world from $\{t_1, t_2, ..., t_i\}$ has exactly r tuples [138]. Then, probability of a tuple t_i being ranked at r is given as,

$$Pr(t_i, r) = p_i S_{i-1,r-1} \tag{6.3}$$

[1] $Pr(t_i, r) = 0$, for $r > i$.

In the above equation,

$$S_{i,r} = \begin{cases} p_i S_{i-1,r-1} + (1-p_i) S_{i-1,r} & \text{if } i \geq r > 0 \\ 1 & \text{if } i = r = 0 \\ 0 & \text{otherwise.} \end{cases}$$

Using recursion for $S_{i,r}$ and equation 6.2, 6.3,

$$\begin{aligned} \Upsilon(t_i) &= \sum_r \alpha^{r-1} Pr(t_i, r) = \sum_r \alpha^{r-1} p_i S_{i-1,r-1} \\ \frac{\Upsilon(t_i)}{p_i} &= \sum_r \alpha^{r-1} S_{i-1,r-1} = \sum_r \alpha^r S_{i-1,r} \\ \frac{\Upsilon(t_{i+1})}{p_{i+1}} &= \sum_r \alpha^r S_{i,r} \\ &= \sum_r \alpha^r (p_i S_{i-1,r-1} + (1-p_i) S_{i-1,r}) \\ &= \alpha p_i \sum_r \alpha^{r-1} S_{i-1,r-1} + (1-p_i) \sum_r \alpha^r S_{i-1,r} \\ &= (1-(1-\alpha)p_i) \Upsilon(t_i)/p_i \end{aligned}$$

We have the base case, $\Upsilon(t_1) = p_1$. Therefore,

$$\Upsilon(t_i) = p_i \prod_{j<i} (1-(1-\alpha)p_j) \tag{6.4}$$

Contribution of a tuple t_i towards global ranking over T can now be analyzed as follows: Tuple t_i contributes $m_i = p_i$ for the computation of its own `rank-score` and contributes $c_i = 1-(1-\alpha)p_i$ of computing `rank-score` for all tuples having score less than its own score.

Theorem 6.1. When all tuples in T are independent, `rank-score` of a tuple t_i can be computed as follows, where $m_i = p_i$ and $c_j = 1-(1-\alpha)p_j$.

$$\Upsilon(t_i) = m_i \prod_{j<i} c_j \tag{6.5}$$

Answering top-1 query: We use a divide and conquer approach for answering top-1 query on T, which forms the basis for our data structure in later section. Let the given relation $T = \{t_1, t_2, ..., t_n\}$ be partitioned into sub-reltations $T_l = \{t_1, t_2, ..., t_{\lceil n/2 \rceil}\}$ and $T_r = \{t_{\lceil n/2 \rceil+1}, t_{\lceil n/2 \rceil+2}, ..., t_n\}$. Also let t^l and t^r represent the top-1 answer for T_l and T_r with `rank-scores` $\Upsilon_{T_l}(t^l)$ and $\Upsilon_{T_r}(t^r)$ respectively, where $\Upsilon_{T_l}(t^l)$ is computed by considering only those tuples $t_j \in T_l$ and $\Upsilon_{T_r}(t^r)$ is computed by considering only those tuples $t_j \in T_r$. Therefore, for $t_i \in T_l$, $\Upsilon_{T_l}(t_i) = m_i \prod_{j<i, t_j \in T_l} c_j$ and similarly for $t_i \in T_r$, $\Upsilon_{T_r}(t_i) = m_i \prod_{j<i, t_j \in T_r} c_j$. Now when both the relations T_l and T_r are merged to form T, we make the following observations:

- The contribution of each tuple towards its own `rank-score` remains unchanged.
- Since all the tuples in T_r have a lower score value than any tuple $t_i \in T_l$ they do not contribute towards the `rank-score` value of t_i computed over entire relation T. Thus $\Upsilon(t_i) = \Upsilon_{T_l}(t_i)$. Hence, t^l still has the highest `rank-score` value $\Upsilon(t^l)$ among the tuples in T_l.
- Since all the tuples in T_l have higher score value than any tuple $t_i \in T_r$, each $t_j \in T_l$ contributes $1 - (1 - \alpha)p_j$ towards `rank-score` value of t_i computed over entire relation T. Let $C_l = \prod_{t_j \in T_l} c_j = \prod_{t_j \in T_l} 1 - (1 - \alpha)p_j$ represents overall contribution of sub-relation T_l. Then $\Upsilon(t_i) = C_l \Upsilon_{T_r}(t_i)$. Since `rank-score` value of every tuple $t_i \in T_r$ gets scaled by the same factor C_l, t^r still has the highest `rank-score` value $\Upsilon(t^r)$ among the tuples in T_r.

Therefore, the top-1 answer over uncertain relation T can be chosen from t^l and t^r based on the their `rank-score` values computed over the entire relation.

6.4.2 Supporting Correlations

If tuple t_i has some preceding alternatives, then equation 6.4 cannot be used to compute its `rank-score` since the event that t_i appears at a position r in a possible world, is no longer independent of the event that exactly $r - 1$ tuples appear in $\{t_1, t_2, ..., t_{i-1}\}$, as in equation 6.3. To overcome this difficulty, we convert the relation T to $\bar{T}^i$ where all the tuples are independent [138]. For any tuple t_i, let τ^i be the pruned version of τ such that it consists of all tuples from τ that have higher score value than that of t_i i.e., $\tau^i = \{t_j | t_j \in \tau, j < i\}$. For example, let $T = \{\tau_1, \tau_2, \tau_3\}$

where, $\tau_1 = \{t_1, t_3, t_6\}, \tau_2 = \{t_2, t_7\}$ and $\tau_3 = \{t_4, t_5\}$ then $\tau_1^5 = \{t_1, t_3\}$, $\tau_2^5 = \{t_2\}$ and $\tau_3^5 = \{t_4\}$. Now for each x-tuple $\tau \in T$, we create an x-tuple $\bar{\tau} = \{\bar{t}\}$ in $\bar{T}^i$ such that:

$$Pr(\bar{\tau}) = Pr(\bar{t}) = \begin{cases} Pr(\tau^i) & \text{if } \tau \neq \tau(t_i) \\ Pr(t_i) & \text{otherwise.} \end{cases}$$

This conversion takes into account the fact that only tuples with a score higher than that of t_i contribute to $Pr(t_i, r)$ as well as to $\Upsilon(t_i)$, and the presence of t_i implies absence of all its related tuples. Combining related tuples into a representative tuple $\bar{t}$ does not affect $\Upsilon(t_i)$ here, since the probability that $\bar{t}$ appears is the same as the probability that any one tuple in $\tau \in T$ with score higher than $score(t_i)$ appears. In other words, $\Upsilon(t_i)$ computed using transformed relation $\bar{T}^i$ is same as $\Upsilon(t_i)$ computed using original relation T. However as all the tuples in $\bar{T}^i$ are independent among themselves, we can now use equation 6.4 on $\bar{T}^i$ to compute the `rank-score` of tuple t_i. Therefore,

$$\Upsilon(t_i) = p_i \prod_{\substack{\bar{t} \in \bar{T}^i \\ \bar{\tau}(\bar{t}) \neq \tau(t_i)}} (1 - (1-\alpha) Pr(\bar{t})) = p_i \prod_{\substack{\tau \in T \\ \tau \neq \tau(t_i)}} (1 - (1-\alpha) Pr(\tau^i)) \tag{6.6}$$

Now we analyze the contribution of an x-tuple towards global ranking over T using the above formula as follows:

- x-tuple τ contributes $m_i = p_i$ for computing `rank-score` of a tuple $t_i \in \tau$.
- x-tuple τ contributes $c_i = 1 - (1-\alpha) Pr(\tau^i)$ for computing `rank-score` of a tuple $t_i \notin \tau$.

Answering top-1 query: Again, we attempt to use a divide and conquer algorithm for answering top-1 query on T by partitioning relation $T = \{t_1, t_2, ..., t_n\}$ into sub-relations $T_l = \{t_1, t_2, ..., t_{\lceil n/2 \rceil}\}$ and $T_r = \{t_{\lceil n/2 \rceil + 1}, t_{\lceil n/2 \rceil + 2}, ..., t_n\}$ and assuming t^l, t^r represent the top-1 answers for T_l, T_r respectively. If property that t^l and t^r remains highest `rank-score` tuples in their respective sub-relations even after merging of T_l and T_r, holds true then reporting top-1 for relation T can be done by simply comparing `rank-score` values of t^l and t^r over entire relation T. Unfortunately, this property may not hold true for t^r.

To illustrate the problem, consider an uncertain relation $T = \{t_1, t_2, t_3, t_4\}$ with $p_1 = 0.35, p_2 = 0.3, p_3 = 0.4, p_4 = 0.45$ and tuples t_2 and t_3 are mutually exclusive. Using equation 6.6, `rank-scores` can be computed as follows ($\alpha = 0.8$):

$\Upsilon(t_1) = 0.35$

$\Upsilon(t_2) = 0.3(1 - 0.2 \times 0.35) = 0.28$

$\Upsilon(t_3) = 0.4(1 - 0.2 \times 0.35) = 0.37$

$\Upsilon(t_4) = 0.45(1 - 0.2 \times 0.35)(1 - 0.2 \times (0.3 + 0.4)) = 0.36$

Top-1 query on T should return tuple t_3 with highest `rank-score` value 0.37. By adopting the divide and conquer approach to tackle the problem, we partition the given relation into $T_l = \{t_1, t_2\}$ and $T_r = \{t_3, t_4\}$. Top-1 query is applied to these sub-relations as follows:

$\Upsilon_{T_l}(t_1) = 0.35$

$\Upsilon_{T_l}(t_2) = 0.3(1 - 0.2 \times 0.35) = 0.28$

$\Upsilon_{T_r}(t_3) = 0.4$

$\Upsilon_{T_r}(t_4) = 0.45(1 - 0.2 \times 0.4) = 0.41$

Thus, t_1 and t_4 will be reported from T_l and T_r as top-1 answers respectively. By simple merge operation, which computes `rank-score` values for t_1, t_4 over relation T and comparing them, t_1 will be reported as top-1 answer for T. However, actual top-1 answer is tuple t_3. The fact that dependance of t_2 and t_3 was ignored while answering top-1 over sub-relation T_r is the root cause behind the disturbance in relative ordering of t_3 and t_4. Therefore in order to maintain the relative ordering of tuples based on their `rank- score` over entire relation during merge, we redefine the expressions for contributions as follows. Here we use the notation $\hat{p}_i$ for sum of probabilities of all tuples t_j which are related to t_i and have score greater than the score of t_i (i.e., $j < i$). In the above example $\hat{p}_3 = p_2 = 0.3$.

$$\hat{p}_i = Pr([\tau(t_i)]^i) = \sum_{\substack{\tau(t_i)=\tau(t_j) \\ j<i}} p_j$$

Now equation 6.6 can be re arranged as follows,

$$\Upsilon(t_i) = \frac{p_i}{(1-(1-\alpha)\hat{p}_i)} \prod_{\tau \in T}(1-(1-\alpha)Pr(\tau^i))$$

$$\frac{\Upsilon(t_i)}{m_i} = \prod_{\tau \in T}(1-(1-\alpha)Pr(\tau^i)) \text{ where, } m_i = \frac{p_i}{(1-(1-\alpha)\hat{p}_i)}$$

$$\frac{\Upsilon(t_{i+1})}{m_{i+1}} = \prod_{\tau \in T}(1-(1-\alpha)Pr(\tau^{i+1}))$$

Here note that $Pr(\tau^i) = Pr(\tau^{i+1})$ for all $\tau \neq \tau(t_i)$. From the above two equations,

$$\left(\frac{\Upsilon(t_{i+1})}{m_{i+1}}\right) / \left(\frac{\Upsilon(t_i)}{m_i}\right) = \frac{1-(1-\alpha)Pr([\tau(t_i)]^{i+1})}{1-(1-\alpha)Pr([\tau(t_i)]^{i})}$$

$$= \frac{1-(1-\alpha)(\hat{p}_i + p_i)}{1-(1-\alpha)\hat{p}_i}$$

$$= c_i$$

The base case is $\Upsilon(t_1) = p_1$. Therefore, we can rewrite equation 6.6 as follows,

$$\frac{\Upsilon(t_{i+1})}{m_{i+1}} = c_i \frac{\Upsilon(t_i)}{m_i} = c_i c_{i-1} \frac{\Upsilon(t_{i-1})}{m_{i-1}} = ... = \prod_{j \leq i} c_j \tag{6.7}$$

Theorem 6.2. For an uncertain relation T, `rank-score` of a tuple t_i can be computed as,

$$\Upsilon(t_i) = m_i \prod_{j<i} c_j$$

where $m_i = \frac{p_i}{(1-(1-\alpha)\hat{p}_i)}$, $c_i = \frac{1-(1-\alpha)(\hat{p}_i+p_i)}{1-(1-\alpha)\hat{p}_i}$ and $\hat{p}_i = \sum t_r$ with tuple t_i and tuple t_r being mutually exclusive such that $r < i$.

This equation is applicable for dependent as well as independent tuples. Note that here m_i and c_i are dependent only on the tuples which are related to t_i, and hence, can be computed/updated efficiently. Moreover, the contribution c_i of a tuple t_i to the `rank-score` of a tuple t_j is the same for all $j > i$. Hence, the relative ordering will not change even if we use our divide and conquer approach. Consider the same example as before. We begin by computing values of m_i and c_i for each tuple.

$m_1 = 0.35$ $m_2 = 0.3$ $m_3 = \frac{0.4}{(1-0.2\times0.3)} = 0.43$ $m_4 = 0.45$

$c_1 = (1 - 0.2 \times 0.35) = 0.93$ $c_2 = (1 - 0.2 \times 0.3) = 0.94$

$c_3 = \frac{(1-0.2\times(0.3+0.4))}{(1-0.2\times0.3)} = 0.91$ $c_4 = (1 - 0.2 \times 0.45) = 0.91$

Now, we partition T into $T_l = \{t_1, t_2\}$, $T_r = \{t_3, t_4\}$ and apply top-1 query to both these sub-relations.

$\Upsilon_{T_l}(t_1) = m_1 = 0.35$ $\Upsilon_{T_l}(t_2) = m_2 \times c_1 = 0.3 \times 0.94 = 0.28$

$\Upsilon_{T_r}(t_3) = m_3 = 0.43$ $\Upsilon_{T_r}(t_4) = m_4 \times c_3 = 0.45 \times 0.91 = 0.41$

It can be seen that t_1 and t_3 are chosen as top-1 from T_l and T_r respectively. During next comparison, t_3 ($\Upsilon(t_3) = m_3 \times c_1 \times c_2 = 0.37$) will be reported as the top-1 tuple, which is correct. Table 6.2 shows m_i and c_i values computed for the uncertain data in Table 6.1.

TABLE 6.2. Calculation of rank-scores of tuples in Table 6.1

$(\alpha = 0.9) : t_1$,$\{t_2, t_4\}$, $\{t_3, t_6\}$, t_5

Tuple	Probability	m	c	Υ
t_1	0.30	0.300	0.970	0.300
t_2	0.40	0.400	0.960	0.388
t_3	0.20	0.200	0.980	0.186
t_4	0.50	0.521	0.948	0.475
t_5	0.30	0.300	0.970	0.260
t_6	0.45	0.459	0.954	0.385

6.5 Proposed Data Structure:

In the earlier sections, we derived the simple closed form expression for calculating $\Upsilon(t_i)$ for a tuple t_i. Now our task is to maintain a dynamic collection of tuples, such that for a given query k, we retrieve top-k `rank-scored` tuples efficiently. We use data structural approach for this problem. Our structure is a balanced binary search tree Δ (e.g. Red black tree, AVL tree) such that each leaf corresponds to a tuple in an uncertain relation T. Moreover, leaves in the tree are sorted in decreasing order of the score i.e., leaves $\ell_1, \ell_2, ..., \ell_n$ of the tree represent tuples $t_1, t_2, ..., t_n$ in the

same order from left to right, such that $score(t_i) > score(t_{i+1})$. Let T_u represents the sub-relation containing tuples associated with leaves of a subtree rooted at node u. i.e., $T_u = \{t_{u'}, t_{u'+1}, ..., t_{u''}\}$ and $\ell_{u'}$ represents the left-most and $\ell_{u''}$ represents the right-most leaf of node u. At each node u, we store a triplet (top_u, M_u, C_u) such that:

- top_u is the tuple (represented by ℓ_{u^*}) with highest `rank-score` among tuples in sub-relation T_u. Here $u' \leq u^* \leq u''$.
- $M_u = m_{u^*} \prod_{u' \leq i < u^*} c_i$ is the contribution of all tuples in T_u towards `rank-score` of top_u.
- $C_u = \prod_{u' \leq i \leq u''} c_i$ is the contribution of all tuples in T_u towards `rank-score` of tuple t_i such that $i > u''$, where $\ell_{u''}$ is the right-most leaf of the subtree rooted at node u.

Since our data structure stores only a constant number of information at each node, and the number of nodes are bounded by $O(n)$, the total space requirement of our data structure is $O(n)$. If node u is a leaf node representing the tuple t_i, then $M_u = m_i, top_u = t_i$ and $C_u = c_i$. If u is an internal node, this information can be computed using the `MERGE` operation given below. Figure 6.1 shows an example for the uncertain data in Table 6.2.

Algorithm 2 MERGE(u)

$v = left - child(u)$, $w = right - child(u)$
if $M_v > C_v \times M_w$ **then**
  $top_u = top_v$
else
  $top_u = top_w$
end if
M_u = max $(M_v, C_v \times M_w)$
$C_u = C_v \times C_w$

Theorem 6.3. The data structure Δ maintains a dynamic collection of tuples such that top-1 tuple, $t^1 = top_{root}$ and $\Upsilon(t^1) = M_{root}$.

Proof. Let t_a be the actual top-1 and $top_{root} \neq t_a$. Let u be the closest node from root, such that $top_u = t_a$, that means $top_{parent(u)} = t_b \neq t_a$. This is because during the merge operation at $parent(u)$, $m_a \prod_{x \leq i < a} c_i < m_b \prod_{x \leq i < b} c_i$, where ℓ_x is the leftmost leaf of $parent(u)$. Multiplying both the sides of the equation with $\prod_{i<x} c_i$, we get $\Upsilon(t_a) < \Upsilon(t_b)$, which is a contradiction to the

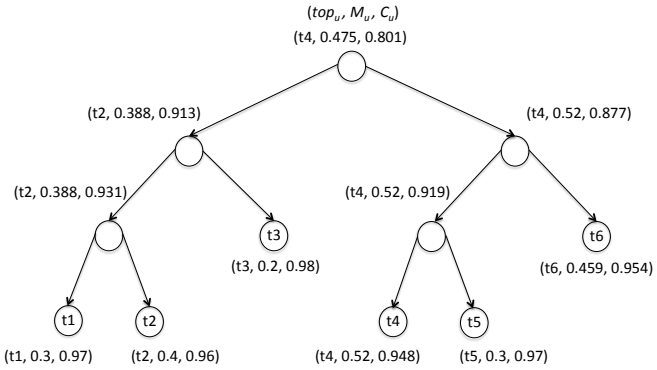

FIGURE 6.1. Data structure for uncertain database in Table 6.1

statement that t_a is the highest `rank-scored` tuple. Therefore $t^1(= t_a)$ will always be at the root and $M_{root} = m_a \prod_{1 \leq i < a} c_i = \Upsilon(t_a) = \Upsilon(t^1)$. □

In the following subsections, we show how to perform different operations such as `update-leaf`, `insert-leaf` and `delete-leaf` on this tree. Later, we use these operations for retrieving top-k tuples, insertion and deletion of tuples.

6.5.1 Update-Leaf

The values m_i and c_i within a leaf node ℓ_i can be changed in constant time. But this will change the m and c values at all nodes which are in the path from ℓ_i to root. Therefore, we need to perform `MERGE` operation on all nodes in the path from ℓ_i to root, starting from $parent(\ell_i)$. Since the height of a balanced binary tree is bounded by $O(\log n)$, the total time for `update-leaf` can also be bounded by $O(\log n)$.

Theorem 6.4. The m_i and c_i values of a leaf can be updated in $O(\log n)$ time.

6.5.2 Insert-Leaf and Delete-Leaf

We first explain, how a one-to-one correspondence between tree leaves and tuples in relation T can be maintained during insertion or deletion of a leaf. To insert a new leaf, we begin by carrying out standard insert procedure of a binary search tree, which would create a new leaf node v. Let w be the parent of this newly created node. Node w being the leaf prior to insertion of v, represents a single tuple from T and should remain as a leaf after insertion of v as well. This can be achieved by

creating a new internal node u, which becomes the parent of v and w. If deletion of a node results in an internal node with only one child, we perform recursive delete on that internal node.

After insert or delete of a leaf node ℓ_i, we need to update the M and C values at each node along the path of insertion or deletion. This can be achieved by performing `MERGE` operation in bottom-up fashion beginning with $parent(\ell_i)$. If tree goes out of balance after insert or delete, necessary rebalancing may force further re-computation at nodes whose left or right subtree is changed. However, such nodes are bounded by the height $O(\log n)$ of the tree. Hence, `Insert-leaf` and `Delete-leaf` operations can be done $O(\log n)$ time.

6.5.3 Retrieving Top-k tuples

In theorem 3, we proved that, by `MERGE` operation the top-1 tuple t^1 will be propagated to root node as top_{root}. Therefore, t^1 can be retrieved in constant time. In order to retrieve the top-2 tuple t^2, we use the following strategy. After retrieving t^1, we set $\Upsilon(t^1) = 0$. As a result, the next highest `rank-scored` tuple t^2 will be propagated as top_{root} instead of t^1. This can be achieved by performing `Update-leaf` operation on leaf ℓ_j (leaf representing the current $top_{root} = t_j$), with it m_j value set to zero. As c_j remains unchanged, update operation affects only the computation of `rank-score` of t_j leaving `rank-score` of all other tuples unchanged. Repeating the same process, we can retrieve top-k tuples with highest `rank-score` values. We can revert back these changes done in data structure by restoring the m values for k retrieved tuples using `Update-leaf` operation. Figure 6.2 shows an example for retrieving top-2 tuple from the uncertain data in Table 6.1.

Theorem 6.5. Top-k `rank-scored` tuples can be retrieved in $O(k \log n)$ time.

Proof. For every tuple t_j retrieved for answering top-k query, we perform `Update-leaf` operation twice: once for setting $m_j = 0$ so that tuple with next highest `rank-score` can be retrieved and next after reporting top-k answers so as to restore the tree changes. Since `Update-leaf` is a $O(\log n)$ time operation, total time for top-k retrieval can be bounded by $O(k \log n)$. □

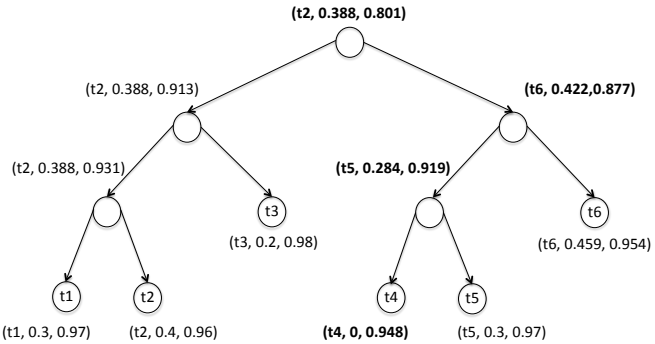

FIGURE 6.2. Data structure in Figure 6.1 after setting $m_4 = 0$ for retrieving top-2

6.5.4 Insert-Tuple and Delete-Tuple

Whenever a tuple t_i gets inserted (deleted) from relation T, we modify our data structure as follows. We begin by carrying out `Insert-leaf` or `Delete-leaf` operation as necessary. If t_i is an independent tuple then at this point all nodes in the tree Δ have correct values for C and M. Hence no further action is necessary. If t_i is not independent, then its insertion(deletion) will change m_j and c_j values for all leaf nodes corresponding to tuples t_j such that $j > i$ and $\tau(t_i) = \tau(t_j)$. These changes can be accommodated by performing `Update-leaf` operation on each ℓ_j.

Figure 6.3 shows an example of inserting a tuple t^* ($score(t_2) > score(t^*) > score(t_3)$) that is mutually exclusive with t_5 in the uncertain data in Table 6.2 whereas Figure 6.4 shows an example for deletion of a tuple. Thus, insertion (deletion) of a tuple can result in one `Insert-leaf` or `Delete-leaf` operation and at maximum $|\tau(t_i)|$ `Update-leaf` operations. Since any x-tuple can have only constant number of tuples, tuple insertion (deletion) can be handled in $O(\log n)$ time. We note that update of a tuple can be simulated by first deleting the tuple and then reinserting the same with updated values.

We summarize the space requirement and performance of the proposed data structure in the following theorem.

Theorem 6.6. A collection of uncertain data (n tuples) can be maintained using a linear size dynamic data structure, which can retrieve top-k `rank-scored` tuples in $O(k \log n)$ time, and can support insertion or deletion of a tuple t in $O(d \log n)$ time, where d is the number of tuples which are related to t.

6.6 Experimental Study

In this section, we present an experimental study with both synthetic and real data evaluating effectiveness of the data structure in handling changes in underlying database and answering top-k queries. All experiments were conducted on 2.4 GHz Intel Core 2 Duo machine with 2GB memory running MAC OS 10.6.4.

Datasets: We created a synthetic dataset containing 100,000 tuples. Score of a each tuple is chosen uniformly at random from [0,100000] and it's probability is uniformly distributed in $(0.5 \times 10^{-5}, 1.5 \times 10^{-5})$. The number of tuples involved in each x-tuple follows the uniform distribution (2,10). Along with synthetic datasets, we also use International Ice Patrol (IIP) Iceberg Sighting Database [1]. Each sighting record in the database contains date, location, number of days the iceberg has drifted, etc. As it is crucial to detect the icebergs drifting for long periods, we use the *number of days drifted* as ranking score. The sighting record also contains a confidence-level attribute according to the source of sighting: R/V (radar and visual), VIS (visual only), RAD (radar only), SAT-LOW (low earth orbit satellite), SAT-MED (medium earth orbit satellite), SAT-HIGH (high earth orbit satellite), and EST (estimated). We converted these seven confidence levels into probabilities 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, and 0.4 respectively. We gathered all records from 1981 to 1991 and 1998 to 2004. Based on it then we created 100,000 tuples dataset by random selection.

[1] http://nsidc.org/data/g00807.html

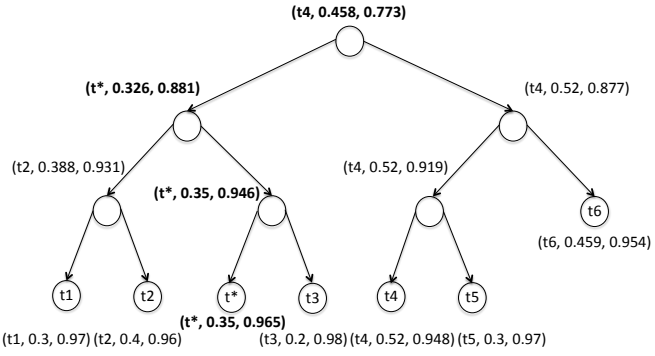

FIGURE 6.3. Data structure in Figure 6.1 after inserting t*

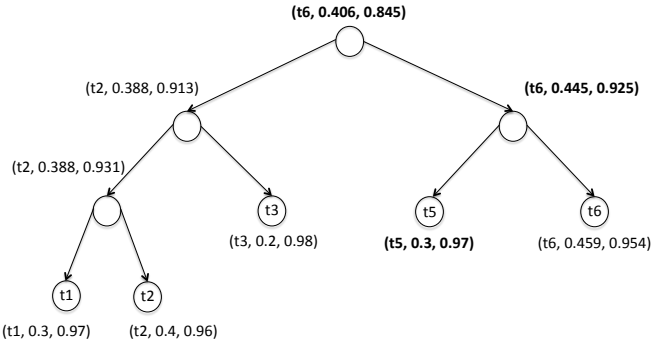

FIGURE 6.4. Data structure in Figure 6.1 after deleting t_4

Results: Experiments in [90] illustrate the effectiveness of ranking function $PRF^e(\alpha)$ at approximating other ranking functions for varying values of α ($\alpha = 1 - 0.9^i, 0 \leq i \leq 200$), where normalized Kendall distance [40] is used to evaluate closeness between the top-100 answers computed using a specific ranking function and $PRF^e(\alpha)$. As revealed by these experiments, ranking functions `U-kRanks, PT-k` are best approximated by $PRF^e(\alpha)$ for $i \approx 50$, hence we choose $\alpha = 1 - 0.9^{50}$ for all of our experiments. Choice of α only determines the quality of approximation and does not affect the query performance of our data structure.

We begin by evaluating the query performance of the data structure. We retrieve top-k tuples from both the datasets for k ranging from 10 to 100. Linear dependance of query time as obtained in the time bounds is evident from the results show in Figure 6.5. Also we can note that, correlations among tuples does not affect the query time of our data structure.

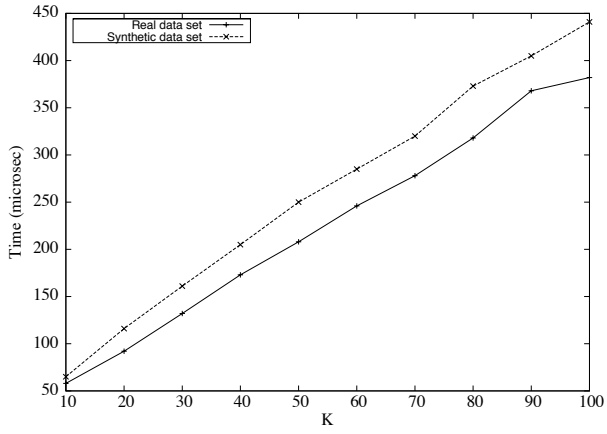

FIGURE 6.5. Top-k query performance

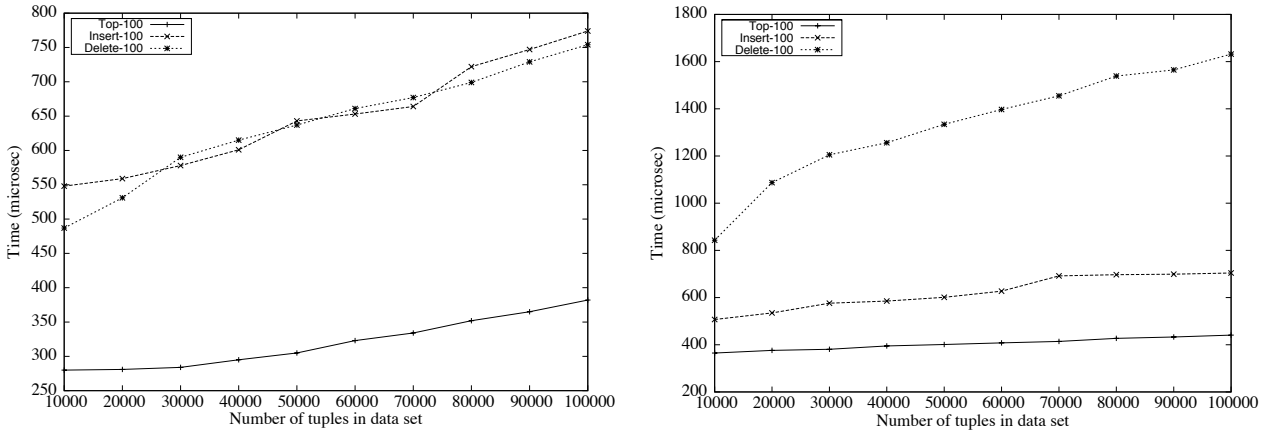

FIGURE 6.6. Processing (insert, delete, top-k) cost on (a) real dataset (b) synthetic dataset

Next set of experiments conducted shows efficiency of our data structure in handling tuple insertions and deletions. Time required for inserting and deleting 100 tuples is measured for datasets of varying sizes. Figure 6.6 (a) and (b) shows that processing time per tuple increases slowly with data size. Whenever a tuple is inserted or deleted, to maintain the correctness of data structure, we also need to update information for leaves corresponding to its related tuples. As all tuples in real data set are assumed to be independent, average insertion/deletion time of a tuple is less than in case of synthetic data having correlations. For synthetic dataset, an x-tuple is selected at random to which a new tuple is added or from which a existing tuple is deleted. We ensure the x-tuple probability to be less than 1 to which a new tuple is being inserted. Position of a new tuple to be inserted in score-sorted ordering of tuples is selected at random whereas tuple to be deleted is always the highest scored tuple in the victim x-tuple. This results in more number of `Update-leaf` operations per tuple deleted than for tuple inserted and its effect on tuple insertion/deletion can be seen from figure 6.6 (b).

The proposed data structure can also be used when data arrives in streaming fashion. Jin et al. [79] have studied the problem of answering top-k queries on sliding windows. Our data structure achieves performance comparable to synopses proposed by them in terms of handling tuple insertion and deletions. Even though our data structure takes linear size as compared to these space efficient

synopses, it can be noted that they rely on random order stream model used in streams algorithm community [23, 24, 56] and in worst case would take linear size as well.

6.7 Related Work

Uncertain data management has attracted a lot of attention in recent years due to an increase in the number of application domains that naturally generate uncertain data. These include sensor networks [38], data cleaning [61] and data integration [50, 26]. Several probabilistic data models have been proposed to capture data uncertainty (e.g TRIO [133], MYSTIQ [35], MayBMS [68], ORION [31], PrDB [121]). Virtually all models have adopted possible worlds semantics. Each data model captures either tuple uncertainty, or attribute uncertainty or both. Further distinction can be made among these models based on support for correlations. Most of the work in probabilistic databases has either assumed independence or support restricted correlations, mutual exclusion being the most common. Recently proposed approaches [121, 83] extend the support for any arbitrary correlations.

Efforts have been made in recent times to extend the semantics of "top-k" to uncertain databases. Soliman et al. [125] defined the problem of ranking over uncertain databases. They proposed two ranking functions, namely `U-Top`k and `U-`k`Ranks`, and proposed algorithms for each of them. Improved algorithms for the same ranking functions were presented later by Yi et al. [138]. Hua et al. [67] proposed another top-k definition `PT-`k (*probabilistic threshold queries*) and proposed efficient solutions. Cormode et al. [32] defined number of key properties satisfied by "top-k" over deterministic data including exact-k, containment, unique-rank, value-invariance, and stability. With each of the existing top-k definition lacking one or more of these properties, Cormode at al. [32] proposed yet another ranking function `expected- rank`. As the list of top-k definitions continued to grow, Li et al. [90] argued that a single specific ranking function may not be appropriate to rank different uncertain databases and empirically illustrated the diverse, conflicting nature of parameterized ranking functions that generalize or can approximate many known ranking functions.

With most of the work for top-k query processing being focused on "one-shot" top-k query for static uncertain data, Chen and Yi [30] were the first to address the dynamic aspect of uncertain

data. They proposed a dynamic data structure to support arbitrary insertions and deletions. For an uncertain relation with n tuples, the structure of [30] answers top-k queries in $O(k + \log n)$ time, handles an update in $O(k \log k \log n)$ time and takes $O(n)$ space. However, this structure is tied to a single ranking function i.e., U-Top$k$ and works only for independent tuples. Moreover, it can be built for some fixed k value and cannot answer a top-j for $j > k$. Dependance of time, required for handling update, on k is also not desirable. Recently, Jin et al. [79] proposed a framework for sliding window top-k queries on uncertain streams supporting several ranking functions. This framework assumes random-order stream model which significantly reduces the space requirement as compared to the worst-case scenario in which any data structure will have to remember every tuple in the current window.

6.8 Summary

Top-k queries over uncertain relation T return a set of the k "best" tuples. Many algorithmic solutions have been proposed for computing top-k answers on a fixed relation T. Thus, any change in the data forces re-computation of top-k answers. With query time of algorithmic solutions being linear to the size of a relation at best, recomputing top-k answers may not be feasible. In this chapter we consider the dynamic problem, that is, how to maintain the top-k query answer when T changes, including tuple insertion and deletions, changes in the probability or score of the tuple. We present a fully dynamic linear space data structure that handles an update in $O(\log n)$ time, and answers a top-k query in $O(k \log n)$ time.

Chapter 7
Similarity Joins for Uncertain Strings

7.1 Introduction

Strings form a fundamental data type in computer systems and string searching has been extensively studied since the inception of computer science. String similarity search takes a set of strings and a query string as input, and outputs all the strings in the set that are similar to the query string. A join extends the notion of similarity search further and require all similar string pairs between two input string sets to be reported. Both similarity search and similarity join are central to many applications such as data integration and cleaning. Edit distance is the most commonly used similarity measure for strings. The edit distance between two strings r and s, denoted by $ed(r, s)$, is the minimum number of single-character edit operations (insertion, deletion, and substitution) needed to transform r to s. Edit distance based string similarity search and join has been extensively studied in the literature for deterministic strings [53, 27, 7, 84, 136, 42]. However, due to the large number of applications where uncertainty or imprecision in values is either inherent or desirable, recent years have witnessed increasing attention devoted to managing uncertain data. Several probabilistic database management systems (PDBMS), which can represent and manage data with explicit probabilistic models of uncertainty, have been proposed to date [133, 124]. Imprecision in data introduces many challenges for similarity search and join in databases with probabilistic string attributes, which is the focus of this paper.

Uncertainty model: Analogous to the models of uncertain databases, two models - string-level and character-level - have been proposed recently by Jeffrey Jestes et al. [77] for uncertain strings. In the string-level uncertanity model all possible instances for the uncertain string are explicitly listed to form a probability distribution function (pdf). In contrast, the character-level model describes distributions over all characters in the alphabet for each uncertain position in the string. We focus on the character-level model as it is realistic and concise in representing the string uncertainty.

Let $\Sigma = \{c_1, c_2, ..., c_\sigma\}$ be the alphabet. A character-level uncertain string is $S = S[1]S[2]...S[l]$, where $S[i]$ $(1 \leq i \leq l)$ is a random variable with discrete distribution over Σ i.e., $S[i]$ is a set of pairs $(c_j, p_i(c_j))$, where $c_j \in \Sigma$ and $p_i(c_j)$ is the probability of having symbol c_j at position i. Formally $S[i] = \{(c_j, p_i(c_j))|c_j \neq c_m \text{ for } j \neq m, \text{ and } \sum_j p_i(c_j) = 1\}$. When the context of a string is unclear we represent $p_i(c_j)$ for string S by $Pr(S[i] = c_j)$. Throughout we use a lower case character to represent a deterministic string (s) against the uncertain string denoted by a upper case character (S). Let $|S|$ ($|s|$) be the length of string S (s). Then the possible worlds of S is a set of all possible instances s of S with probability $p(s)$, $\sum p(s) = 1$. S being a character-level uncertain string, $|S| = |s|$ for any of its possible instances.

Query semantics: In addition to capturing uncertainty in the data, one must define the semantics of queries over the data. In this regard, a powerful model of possible-world semantics has been the backbone of analyzing the correctness of database operations on uncertain data. For uncertain string attributes, Jestes et al. [77] made the first attempt to extend the notion of similarity. They used expected edit distance (*eed*) over all possible worlds of two uncertain strings. Given strings R and S, $eed(R, S) = \sum_{r_i, s_j} p(r_i)p(s_j)ed(r_i, s_j)$, where s_j (r_i) is an instance of S (R) with probability $p(s_j)$ ($p(r_i)$). Though *eed* seems like a natural extension of edit distance as a measure of similarity, it has been shown that it does not implement the possible-world semantics completely at the query level [51]. Consider a similarity search query on a collection of deterministic strings with input string r. Then, string s is an output only if $ed(r, s) \leq k$. For such a query R over an uncertain string collection, possible world semantics dictate that we apply the same predicate $ed(r, s) \leq k$ for each possible instance r of R, s of S and aggregate this over all worlds. Thus, a possible world with instances r, s can contribute in deciding whether S is similar to R only if s is within the desired edit distance of r. However, for the *eed* measure, all possible worlds (irrespective but weighted by edit distance) contribute towards the overall score that determines the similarity of S with R. To overcome this problem, in [51] the authors have proposed a (k, τ)-matching semantic scheme. Using this semantic, given a edit distance threshold k and probability threshold τ, R is similar to S if $Pr(ed(R, S) \leq k) > \tau$. We use this similarity definition in this paper for answering join queries.

Problem definition: Given two sets of uncertain strings $\mathcal{R}$ and $\mathcal{S}$, an edit-distance threshold k and a probability threshold τ, similarity join finds all similar string pairs $(R, S) \in \mathcal{R} \times \mathcal{S}$ such that $Pr(ed(R, S) \leq k) > \tau$. Without loss of generality, we focus on self join in this paper i.e., $\mathcal{R} = \mathcal{S}$.

Related work: Uncertain/Probabilistic strings have been the subject of study for the past several years. Efficient algorithms and data structures are known for the problem of string searching in uncertain text [70, 5, 71, 139]. In [51] authors have studied the approximate substring matching problem, where the goal is to report the positions of all substrings of uncertain text that are similar to the query string. Recently, the problem of similarity search on a collection of uncertain strings has been addressed in [34]. However, most of these works support only deterministic strings as query input. Utilizing these techniques for uncertain string as input would invariably need all its possible worlds to be enumerated, which may not be feasible to do taking into account the resultant exponential blowup in query cost. Though the problem of similarity join on uncertain strings has been studied in [77], it makes use of expected edit distance as a measure of similarity. We make an attempt to address some of the challenges involved in uncertain string processing by investigating similarity joins on them in this paper.

7.2 Preliminaries

In this section we briefly review filtering techniques for deterministic strings available in literature and extend them for uncertain strings later in the article. Let r, s be the two deterministic strings and k be the edit distance threshold.

7.2.1 q-gram Filtering

We partition s it into $k + 1$ disjoint segments $s^1, s^2, ..., s^{k+1}$. For simplicity let each segment is of length $q \geq 1$ i.e, $s^x = s[((x - 1)q + 1)..xq]$. Further, let $pos(s^x)$ represents the starting position of segment s^x in string s i.e, $pos(s^x) = (x - 1)q + 1$. Then using a pigeonhole principle, if r is similar to a string s, it should contain a substring that matches a segment in s. A straightforward method to achieve this is to obtain a set $q(r)$ enumerating all substrings of r of length q and for each substring check whether it matches s^x for $x = 1, 2, .., k + 1$. However, we do not need to

consider all substrings of r. In [89] authors have shown that we can obtain a set $q(r,x) \subseteq q(r)$ for each segment of s such that it is sufficient to test each substring $w \in q(r,x)$ for a match with s^x. Table 7.1 shows sets $q(r,x)$ populated for a sample string r. The substring selection proposed in [89] is guided by following observations:

- Shift-based selection: Let w be the substring of r with start position smaller than $(pos(s^x) - k)$ or larger than $(pos(s^x) + k)$. Then even if w matches segment s^x, strings r and s cannot be similar based on such an alignment. Hence, we do not need to include such w in set $q(r,x)$.
- Position aware selection: This tightens the number of substrings that can be included in set $q(r,x)$ by taking into account the length difference of strings r and s i.e., $\Delta = abs(|r| - |s|)$. A substring w of r with start position smaller than $(pos(s^x) - \lfloor (k-\Delta)/2 \rfloor)$ or larger than $(pos(s^x) + \lfloor (k+\Delta)/2 \rfloor)$, even if matched with s^x, cannot lead to an alignment of r and s that has edit distance between them within desired threshold k.

Therefore, set $q(r,x)$ includes substrings of r with start positions in the range $\lceil pos(s^x) - \lfloor (k-\Delta)/2 \rfloor, pos(s^x) + \lfloor (k+\Delta)/2 \rfloor \rceil$ and with length q. Number of substrings in set $q(r,x)$ is thus bounded by $k+1$. In [89] authors prove that the substring selection satisfy "completeness" ensuring any similar pair (r,s) will be found as a candidate pair. Please refer to the article [89] for more details. We use a generalization of this filtering technique by partitioning s into $m > k$ partitions. As a consequence, for a string r to be similar to s, it should contain substrings matching more segments of s [89, 108]. Following lemma summarizes this result.

Lemma 7.1. Given a string r and s, with s partitioned into $m > k$ disjoint segments, if r is similar to s within an edit threshold k, r must contain substrings that match at-least $(m-k)$ segments of s.

Once again by assuming each segment of s to be of length $q \geq 1$, we can compute the set $q(r,x)$ and attempt to match each $w \in q(r,x)$ with s^x as before to apply the above lemma.

7.2.2 Frequency Distance Filtering

The intuition behind this filtering is that if two strings are similar, then the frequency of the alphabet symbols in two strings should also be similar [80]. Given a string s from the alphabet

Σ, frequency vector $f(s)$ is defined as $f(s) = [f(s)_1, f(s)_2, ..., f(s)_\sigma]$, where $f(s)_i$ is the count of ith alphabet of Σ i.e, c_i. Let $f(r)$ and $f(s)$ be the frequency vectors of r and s respectively. Then frequency distance of r and s is defined as $fd(r, s) = max\{posD, negD\}$. Frequency distance provides a lower bound for edit distance between r and s i.e., $fd(r, s) \leq ed(r, s)$ and can be computed efficiently [80]. Thus, we can safely decide that r is not similar to s if $fd(r, s) > k$.

$$posD = \sum_{f(r)_i > f(s)_i} f(r)_i - f(s)_i, negD = \sum_{f(r)_i < f(s)_i} f(s)_i - f(r)_i$$

7.3 q-gram Filtering

In this section we adopt and extend the ideas introduced for deterministic strings earlier in Section 7.2.1 to uncertain strings. We begin with the simpler case where either of the two uncertain strings R and S is deterministic. Let R be that string with r being its only possible instance. We try to achieve an upper bound on the probability of r and S being similar i.e., $Pr(ed(r, S) \leq k)$. We then build upon this result for the case when both strings are uncertain and obtain an upper bound on the probability of R and S being similar i.e., $Pr(ed(R, S) \leq k)$.

Before proceeding, we introduce some notation and definitions. A string w of length l matches a substring in T starting at position i with probability $Pr(w = T[i..i+l-1]) = \prod_{ps=1}^{l} p_{i+ps-1}(w[ps])$. A string w matches T with probability $Pr(w = T) = \prod_{ps=1}^{l} p_{ps}(w[ps])$ if $|w| = |T| = l$; otherwise it is 0. We simply say w matches with T (or vice versa) if $Pr(w = T) > 0$. The probability of string W matching T is given by $Pr(W = T) = \prod_{ps=1}^{l} \sum_{c_j \in \Sigma} Pr(W[ps] = c_j) \times Pr(T[ps] = c_j)$. Once again, we say W matches T if $Pr(W = T) > 0$ for simplicity.

7.3.1 Bounding $Pr(ed(r, S) \leq k)$

The possible worlds Ω of S is the set of all possible instances of S. A possible world $pw_j \in \Omega$ is a pair $(s_j, p(s_j))$, where s_j is an instance of S with probability $p(s_j)$. Let $p(pw_j) = p(s_j)$ denote the probability of existence of a possible world pw_j. Note that s_j is a deterministic string and $\sum p(pw_j) = 1$. Then by definition, $Pr(ed(r, S) \leq k) = \sum_{ed(r,s_j) \leq k} p(pw_j)$.

Necessary condition for $Pr(ed(r, S) \leq k) > 0$: We partition the string S into $m > k$ disjoint substrings. For simplicity, let q be the length of each partition. Note that each partition $S^1, S^2, ..., S^m$ is an uncertain string. Let r contain substrings matching $m' \leq m$ segments of S i.e., the number of segments of S with $Pr(w = S^x) > 0$ for any substring w of r is m'. Then it can be seen that for any $pw_j \in \Omega$, r contains substrings that match with at most m' segments from $s_j^1, s_j^2, ..., s_j^m$ that partition s_j. Based on this observation, the following lemma establishes the necessary condition for $Pr(ed(r, S) \leq k) > 0$.

Lemma 7.2. Given a string r and a string S partitioned into $m > k$ disjoint segments, for r to be similar to S i.e., $Pr(ed(r, S) \leq k) > 0$, r must contain substrings that match at-least $(m - k)$ segments of S.

While applying the above lemma, we do not need to consider all substrings of r of length q listed in $q(r)$. We can obtain a set $q(r, x)$ using position aware selection as described earlier and use it to match against segment S^x. Table 7.1 shows the above lemma applied to a collection of uncertain strings. None of the segments of S_1 match any substring in r and hence they can not form a candidate pair. For S_2, even though the second segment matches some substring in r, we do not use it as we know by position aware substring selection that such an alignment can not lead to an instance of S that is similar to r. We can reject S_2 as well since it has only one matched segment. Strings S_3 and S_4 survive this pruning step and are taken forward.

Computing upper bound for $Pr(ed(r, S) \leq k)$: So far we were interested in knowing if there exists a substring $w \in q(r, x)$ that matches segment S^x. We now try to compute the probability that one or more substrings in $q(r, x)$ match S^x. Let E_x denote such an event with probability α_x. Then $\alpha_x = Pr(E_x) = \sum_{w \in q(r,x)} Pr(w = S^x)$. The correctness of α_x relies on the following observations:

- $q(r, x)$, being a set, contains all distinct substrings.
- Event of substring $w_i \in q(r, x)$ matching S^x is independent of substring $w_j \in q(r, x)$ matching S^x for $w_i \neq w_j$.

TABLE 7.1. Application of q-gram filtering

$m = 3$, $q = 2$, $k = 1$, τ=0.25			
r	GGATCC		
$q(r, x)$	GG	GA	TC
	GA	AT	CC
		TC	
S_1	A{(C,0.5),(G,0.5)}A{(C,0.5),(G,0.5)}AC		
	(AC,0.5)	(AC,0.5)	(AC,1)
	(AG,0.5)	(AG,0.5)	
S_2	AA{(G,0.9),(T,0.1)}G{(C,0.3),(G,0.2),(T,0.5)}C		
	(AA,1)	(GG,0.9)	(CC,0.3)✓
		(TG,0.1)	(GC,0.2)
			(TC,0.5)✓
S_3	G{(A,0.8),(G,0.2)}CT{(A,0.8),(C,0.1),(T,0.1)}C		
	(GA,0.8)✓	(CT,1)	(AC,0.8)
	(GG,0.2)✓		(CC,0.1)✓
			(TC,0.1)✓
S_4	{(G,0.8),(T,0.2)}GA{(C,0.3),(G,0.2),(T,0.5)}CT		
	(GG,0.8)✓	(AC,0.3)	(CT,1)
	(TG,0.2)	(AG,0.2)	
		(AT,0.5)✓	

Next, our idea is to prune out the possible worlds of S which can not satisfy the edit-distance threshold k with r and obtain a set $\mathcal{C} \subseteq \Omega$ of candidate worlds. We can then use $Pr(\mathcal{C}) = \sum_{pw_j \in \mathcal{C}} p(pw_j)$ as the upper bound on $Pr(ed(r, S) \leq k)$. Consider a possible world pw_j in which s_j is the possible instance of S. s_j being the deterministic string, we can apply the process of q-gram filtering described in Section 7.2.1 to quickly assess if s_j can give edit distance within threshold k. If yes, pw_j is a candidate world and we include it in $\mathcal{C}$. This naive method requires all possible worlds of S to be instantiated and hence is too expensive to be used. Below we show how to achieve the desired upper bound i.e., $Pr(\mathcal{C})$ without explicitly listing set Ω or $\mathcal{C}$.

For ease of explanation, let $m = k + 1$. We partition the possible worlds in Ω into sets $\Omega_0, \Omega_1, ..., \Omega_m$ such that:

- Ω_y includes any possible world pw_j where r contains substrings matching exactly y segments from $s_j^1, ..., s_j^m$ that partition s_j i.e., $y = |\{s_j^x | s_j^x \in q(r, x) \text{ for } x = 1, 2, .., m\}|$.
- $\Omega = \Omega_0 \cup \Omega_1 \cup \Omega_2 \cup ... \cup \Omega_m$
- $\Omega_y \cap \Omega_z = \emptyset$ for $y \neq z$

With this partitioning of Ω, we have following:

$$Pr(\mathcal{C}) = Pr(\Omega_1 \cup \Omega_2 \cup ... \cup \Omega_m) = Pr(\Omega \setminus \Omega_0)$$
$$= Pr(\Omega) - Pr(\Omega_0) = 1 - \prod_{x=1}^{m}(1 - \alpha_x)$$

In the above equation, Ω_0 denotes the event that none of the segments of S match substrings of r. By slight abuse of notation, we say S^x matches r (using position aware substring selection) if $\alpha_x > 0$. Then, the following lemma summarizes our result on the upper bound.

Lemma 7.3. Let r and S be the given strings with edit threshold k. If S is partitioned into $m = k+1$ disjoint segments, $Pr(ed(r, S) \leq k)$ is upper bounded by $(1 - \prod_{x=1}^{m}(1 - \alpha_x))$, where α_x gives the probability that segment S^x matches r.

Generalizing upper bound for $m > k$: Finally, we turn our attention to compute $Pr(\mathcal{C})$ for the scenario where S is partitioned into $m > k$ segments. Once again considering the partitioning of Ω introduced above $Pr(\mathcal{C}) = Pr(\cup_{y=(m-k)}^{m}\Omega_y) = \sum_{y=m-k}^{m} Pr(\Omega_y)$. Then we observe that computing $Pr(\Omega_y)$ in this equation boils down to the following problem: There are m events E_x ($x = 1, 2, ..m$) and we are given $Pr(E_x) = \alpha_x$. What is the probability that exactly y events (among those m events) happen? Our solution is as follows. Let $Pr(i, j)$ denote the probability that, within the first i events, j of them happen. We then have the following recursive equation: $Pr(i, j) = Pr(E_i)Pr(i-1, j-1) + (1 - Pr(E_i))Pr(i-1, j)$. By populating the $m \times m$ matrix using a dynamic programming algorithm based on the above recursion, we can lookup the last column to find out $Pr(\Omega_y)$ for $y = m - k, ..., m$. This recursion gives us an efficient $(O(m^2))$ way to compute $Pr(\mathcal{C})$. We note that it is possible to improve the running time to $O(m(m-k))$, but leave out the details for simplicity.

Theorem 7.4. Let r and S be the given strings with edit threshold k. Also assume S is partitioned into $m > k$ disjoint segments and α_x represents the probability that segment S^x matches r. Then $Pr(ed(r, S) \leq k)$ is upper bounded by the probability that at-least $(m - k)$ segments of S match r or in another words probability that r contains substrings matching at-least $(m - k)$ segments of S.

Continuing the example in Table 7.1, we now try to apply the above theorem to strings S_3 and S_4. For S_3 we have $\alpha_1 = 1$, $\alpha_2 = 0$, and $\alpha_3 = 0.2$. Therefore the upper bound on S_3's similarity with r is $0.2 < \tau$ and S_3 can be rejected. Even though four out of six possible worlds of S_3 contribute to $\mathcal{C}$, the probability of each of them being small their collective contribution falls short of τ. Similarly the upper bound for S_4 can be computed as 0.4 and the pair (r, S_4) qualifies as a candidate pair. Thus Theorem 7.4 integrates q-gram filtering and probabilistic pruning.

Let string S be preprocessed such that each segment S^x is maintained as a list of pairs $(s_j^x, p(s_j^x))$, where s_j^x is an instance of S^x with probability $p(s_j^x)$. Also assume r is preprocessed and sets $q(r, x)$ are available to us for $x = 1, 2, .., m$ ($|q(r, x)| = k + 1$, $\sum_{x=1}^{m} |q(r, x)| = (k + 1)m$). Then the desired upper bound can be computed efficiently by applying the above theorem as it only adds the following computational overhead in comparison to its counterpart of deterministic strings: (1) computation cost for α_x of each segment is bounded by k and mk overall, (2) the cost of computing $Pr(\mathcal{C})$ using dynamic programming is bounded by $m(m - k)$.

7.3.2 Bounding $Pr(ed(R, S) \leq k)$

In this subsection, we follow the analysis from the earlier subsection taking into account the uncertainty introduced for string R. The possible worlds Ω of R and S is the set of all possible instances of $R \times S$. A possible world $pw_{i,j} \in \Omega$ is a pair $((r_i, s_j), p(r_i) * p(s_j))$, where s_j (r_i) is an instance of S (R) with probability $p(s_j)$ ($p(r_i)$). Also $p(r_i) * p(s_j)$ denote the probability of existence of a possible world $pw_{i,j}$ and $\sum p(pw_{i,j}) = 1$. Then by definition, $Pr(ed(R, S) \leq k) = \sum_{ed(r_i,s_j)\leq k} p(pw_{i,j})$.

Necessary condition for $Pr(ed(R, S) \leq k) > 0$: We begin by partitioning the string S into $m > k$ disjoint substrings as before and assume q to be the length of each partition. Then the following lemma establishes the necessary condition for R to be similar to S within edit threshold.

Lemma 7.5. Given a string R and a string S partitioned into $m > k$ disjoint segments, for R to be similar to S i.e., $Pr(ed(R, S) \leq k) > 0$, R must contain substrings that match at-least $(m - k)$ segments of S.

The correctness of the above lemma can be verified by extending the earlier observation as follows: Let R contain substrings matching $m' \leq m$ segments of S i.e., the number of segments of S with $Pr(W = S^x) > 0$ for any (uncertain) substring W of R is m'. Then for any $pw_{i,j} \in \Omega$, r_i contains substrings that match with at most m' segments from $s_j^1, s_j^2, ..., s_j^m$ that partition s_j. Next, we obtain a set $q(R, x)$ for each segment S^x of S using the position aware substring selection. This allows us to only test substrings $W \in q(R, x)$ for a match against S^x. We highlight that the substring selection mechanism only relies on the length of two strings R and S, start position of a substring W of R and that of S^x. Therefore following same arguments in [89], we can prove that any similar pair (R, S) will be reported as a candidate.

Computing α_x: Let E_x denote an event that one or more substrings in set $q(R, x)$ match segment S^x and let α_x be its probability. Using a trivial extension of the earlier result in Section 7.3.1, we could perhaps compute $\alpha_x = Pr(E_x) = \sum_{W \in q(R,x)} Pr(W = S^x)$. However, we show that this leads to incorrect computation of α_x and requires a careful investigation. Let $R = A\{(A, 0.8), (C, 0.2)\}AATT$, $S = A\{(A, 0.8), (C, 0.2)\}AGCT$, $k = 1$ and $q = 3$. Then, we have $S^1 = A\{(A, 0.8), (C, 0.2)\}A$, $q(R, 1) = \{A\{(A, 0.8), (C, 0.2)\}A,\ \{(A, 0.8), (C, 0.2)\}AA\ \}$. Using the above formula $Pr(E_1) = 0.64 + 0.04 + 0.64 = 1.32$, which is definitely incorrect. To understand the scenario better, let's replace each substring $W \in q(R, x)$ by a list of pairs $(w_j, p(w_j))$, where w_j is an instance of W with probability $p(w_j)$. Note that it is only a different way of representing set $q(R, x)$ and both representations are equivalent. $q(R, 1) = \{(AAA, 0.8), (ACA, 0.2), (AAA, 0.8), (CAA, 0.2)\}$ and $Pr(E_1) = \sum_{w \in q(R,x)} p(w) \times Pr(w = S^x) = 1.32$ as before. However, this representation reveals that we have violated the second observation which requires matching of two substrings $w_i, w_j \in q(R, x)$ with S^x to be independent events. In the current example, both occurrences of a substring AAA in $q(R, 1)$ belong to same possible world and effectively its probability contributes twice to $Pr(E_1)$.

We overcome this issue by obtaining an equivalent set $q(r, x)$ of $q(R, x)$ that satisfies the substring uniqueness requirement i.e., $w_i \neq w_j$ for all $w_i, w_j \in q(r, x)$ with $i \neq j$, and implicitly make the matching of two of its substrings with S^x independent events. To achieve this we pick up

all distinct (deterministic) substrings $w \in q(R, x)$ (think of a representation of set $q(R, x)$ consisting of $(w_j, p(w_j))$ pairs) to be part of $q(r, x)$. To distinguish between these two sets, let $p_R(w_j)$ represent the probability associated with substring w_j in $q(R, x)$ and $p_r(w_j)$ be the same for $q(r, x)$. Then, we maintain the equivalence of sets by following the two step process described below for each $w \in q(r, x)$ and obtain the probability to be associated with it i.e., $p_r(w)$.

1. Sort all occurrences of w in $q(R, x)$ by their start positions in R. Group together all occurrences that overlap with each other in R to obtain groups $g_1, g_2, ...$ Then no two occurrences across the groups overlap each other. Such a grouping is required only when there is a suffix-prefix match for w (i.e., some suffix of w represents same string as its prefix), otherwise all its overlapping occurrences represent different possible worlds of R and hence are in a single group by themselves. We assign the probability $p(g_i)$ to each group g_i as described below. Let ps_j represent the start position of occurrence w_j in R for $j = 1, 2, .., |g_i|$. The region of overlap between an occurrence w_j of w and its previous occurrences in R is given by range $[y, z] = [ps_j, ps_{j-1}+q-1]$. We define $\beta_j = \beta_{j-1}+pr_R(w_j)-Pr(w_j[1..(z-y+1)] = R[y..z])$ with the initial condition $\beta_0 = 1$, $ps_0 = -1$. Then $p(g_i) = \beta_{|g_i|}$. In essence, we keep adding the probability of every occurrence while taking out the probability of its overlap.

2. Assign $p_r(w) = 1 - \prod(1 - p(g_i))$.

The first step combines all overlapping occurrences into a single event and then we find out the probability that at-least one of these events takes place in second step. Now we can correctly compute the probability of event S^x matching substrings in $q(R, x)$ by using its equivalent set $q(r, x)$ as $\alpha_x = Pr(E_x) = \sum_{w \in q(r,x)} p_r(w) \times Pr(w = S^x)$. For the example under consideration, for a substring "AAA" we obtain a single group with its associated probability 0.8 using the process described above. Then $q(r, 1) = \{(AAA, 0.8), (ACA, 0.2), (CAA, 0.2)\}$ and $Pr(E_1) = 0.68$ is correctly computed.

Computing upper bound for $Pr(ed(r, S) \leq k)$: Finally, to obtain the upper bound on $Pr(ed(R, S) \leq k)$ we obtain set $\mathcal{C} \subseteq \Omega$ by pruning out those possible worlds which can not satisfy the edit-distance

threshold k. Consider a possible world $pw_{i,j}$ in which s_j (r_i) is a possible instance of S (R). Both r_i and s_j being deterministic strings, we can quickly assess if r_i and s_j can be within edit distance k by applying the process of q-gram filtering described in Section 7.2.1. If affirmative, $pw_{i,j}$ is a candidate world and we include it in $\mathcal{C}$. However, our goal is to compute $Pr(\mathcal{C})$ without enumerating all possible worlds of $R \times S$. As before, we partition the possible worlds in Ω into sets $\Omega_0, \Omega_1, ..., \Omega_m$ such that $\Omega = \cup_{y=0}^{m} \Omega_y$ and $\Omega_y \cap \Omega_z = \emptyset$ for $y \neq z$. Moreover, Ω_y includes any possible world $pw_{i,j}$ where r_i contains substrings matching exactly y segments from $s_j^1, ..., s_j^m$ that partition s_j i.e, $y = |\{s_j^x | s_j^x \in q(r_i, x) \text{ for } x = 1, 2, .., m\}|$. Then $Pr(\mathcal{C}) = Pr(\cup_{y=(m-k)}^{m} \Omega_y) = \sum_{y=m-k}^{m} Pr(\Omega_y)$ and can be computed by following the same dynamic programming approach described earlier. Therefore the key difference in the current scenario (both R and S are uncertain) from the one in the previous subsection is the computation of α_x. After computing all α_x we can directly apply Lemma 7.2 and Theorem 7.4 and are rewritten as below. By slight abuse of notation as before, we say S^x matches R if $\alpha_x > 0$.

Lemma 7.6. Let R, S be the given strings with edit threshold k. If S is partitioned into $m = k + 1$ disjoint segments, $Pr(ed(R, S) \leq k)$ is upper bounded by $(1 - \prod_{x=1}^{m}(1 - \alpha_x))$, where α_x gives the probability that segment S^x matches R.

Theorem 7.7. Let R, S be the given strings with edit threshold k. Also assume S is partitioned into $m > k$ disjoint segments and α_x represents the probability that segment S^x matches R. Then $Pr(ed(R, S) \leq k)$ is upper bounded by the probability that at-least $(m - k)$ segments of S match R i.e., the probability that R contains substrings matching at-least $(m - k)$ segments of S.

It is evident that the cost of computing the upper bound in the above theorem is dominated by the set $q(r, x)$ computations. If this is assumed to be part of the preprocessing then the overhead involved is exactly the same as in the previous subsection. Let the fraction of uncertain characters in the strings be θ, and the average number of alternatives of an uncertain character be γ. For analysis of q-gram filtering, we assume uncertain character positions to be uniformly distributed from now onwards. Then $|q(r, x)| = (k + 1)\gamma^{\theta \cdot q}$, and computing set $q(r, x)$ for each segment takes $q\gamma^{\theta \cdot q}$ times

when string R is deterministic (previous subsection). Note that the multiplicative q appears only when substring w has a suffix-prefix match and its occurrences in set $q(R, x)$ overlap. Assuming typical values $\theta = 20\%$, $\gamma = 5$ and $q = 3$, it takes only two and half times longer to compute α_x when R is uncertain using $q(r, x)$.

7.4 Indexing

Using Theorem 7.7 we observe that if a string R does not have substrings that match a sufficient number of segments of S, we can prune the pair (R, S). We use an indexing technique that facilitates the implementation of this feature to prune large numbers of dissimilar pairs. So far we assumed each string S is partitioned into m segments, each of which is of length q. In practice, we fix q as a system parameter and then divide S into as many disjoint segments as necessary i.e., $m = max(k + 1, \lfloor |S|/q \rfloor)$. Without loss of generality let $m = \lfloor |S|/q \rfloor$. We use an even-partition scheme [89, 108] so that each segment has a length of q or $q + 1$. Thus we partition S such that the last $|S| - \lfloor |S|/q \rfloor * q$ segments have length $q + 1$ and length is q for the rest of them.

Let $\mathcal{S}_l$ denote the set of strings with length l and $\mathcal{S}_l^x$ denote the set of the x-th segments of strings in S_l. We build an inverted index for each $\mathcal{S}_l^x$ denoted by $\mathcal{L}_l^x$ as follows. Consider a string $S_i \in \mathcal{S}_l$. We instantiate all possibilities of its segment S_i^x and add them to $\mathcal{L}_l^x$ along with their probabilities. Thus $\mathcal{L}_l^x$ is a list of deterministic strings and for each string w, its inverted list $\mathcal{L}_l^x(w)$ is the set of uncertain strings whose x-th segment matches w tagged with probability of such a match. To be precise, $\mathcal{L}_l^x(w)$ is enumeration of pairs $(i, Pr(w = S_i^x))$ where i is the string-id. By design, each such inverted list $\mathcal{L}_l^x(w)$ is sorted by string-ids as described later. We emphasize that a string-id i appears at most once in any $\mathcal{L}_l^x(w)$ and in as many lists $\mathcal{L}_l^x(w)$ as the number of possible instances of S_i^x. We use these inverted indices to answer the similarity join query as follows.

We sort strings based on their lengths in ascending order and visit them in the same order. Consider the current string $R = S_i$. We find strings similar to R among the visited strings only using the inverted indices. This implies we maintain indices only for visited strings to avoid enumerating a string pair twice. It is clear that we need to look for similar strings in $\mathcal{S}_l$ by querying its associated index only if $|R| - k \leq l \leq |R|$. To find strings similar to R, we first obtain candidate strings using

the proposed indexing as described in next paragraph. We then subject these candidate pairs to frequency distance filtering (Section 7.5). Candidate pairs that survive both these steps are evaluated with CDF bounds (Section 7.6.1) with the final verification step (Section 7.6.2) outputting only the strings that are similar to R. After finding similar strings for $R = S_i$, we partition S_i into $m > k$ (as dictated by q) segments and insert the segments into appropriate inverted index. Then we move on to the next string $R = S_{i+1}$ and iteratively find all similar pairs.

Finally, given a string R, we show how to query the index associated with $\mathcal{S}_l$ to find candidate pairs (R, S) such that $S \in \mathcal{S}_l$ and $Pr(ed(R, S) \leq k) > \tau$. We preprocess R to obtain $q(r, x)$ that can be used to query each inverted index $\mathcal{L}_l^x$. For each $w \in q(r, x)$ we obtain an inverted list $\mathcal{L}_l^x(w)$. Since all lists are sorted by string-id, we can scan them in parallel to produce a merged (union) list of all string-ids i along with the α_x computed for each of them. We maintain a top pointer in each $\mathcal{L}_l^x(w)$ list that initially points to its first element. At each step, we find out the minimum string-id i among the elements currently at the top of each list, compute α_x for a pair (R, S_i) using the probabilities associated with string-id i in all $\mathcal{L}_l^x(w)$ lists (if present). After outputting the string-id and its α_x as a pair in the merged list, we increment the top pointers for those $\mathcal{L}_l^x(w)$ lists which have the top currently pointing to the element with string-id i. Let the merged list be $\mathcal{L}\alpha_x$. Once again all $\mathcal{L}\alpha_x$ lists for $x = 1, 2, .., m$ are sorted by string-ids. Therefore by employing top pointers and scanning lists $\mathcal{L}\alpha_x$ in parallel, we can count the number of segments in S_i that matched with their respective $q(r, x)$ by counting the number of $\mathcal{L}\alpha_x$ lists that contain string-id i. If the count is less than $m - k$ we can safely prune out candidate pair (R, S_i) using Lemma 7.6. Otherwise, we can compute the upper bound on $Pr(ed(R, S_i) \leq k)$ by supplying the α_x values already computed to the dynamic programming algorithm. If the upper bound does not meet our probability threshold requirement, we can discard string S_i as it can not be similar to R by Theorem 7.7, otherwise (R, S_i) is a candidate pair.

Given a string R, the proposed indexing scheme allows us to obtain all strings $S \in \mathcal{S}$ that are likely to be similar to R without explicitly comparing R to each and every string in $\mathcal{S}$ as has been done for related problems in the area of uncertain strings [77, 51, 34]. For a string r in a

deterministic strings collection, we need to consider $m(k+1)$ of its substrings while answering the join query using the procedure just described. In comparison, in the probabilistic setting we need to consider $m(k+1)\gamma^{\theta \cdot q}$ deterministic substrings of R. Moreover, a string-id can belong to at most $\gamma^{\theta \cdot q}$ inverted lists in $\mathcal{L}_l^x$ in probabilistic setting whereas inverted lists are disjoint for deterministic strings collection. Thus, the proposed indexing achieves competitive performance against its counterpart for answering a join query over deterministic strings. Further, indexing scheme uses disjoint q-grams of strings instead of overlapping ones as in [51, 34]. This allows us to use slightly larger q with same storage requirements.

7.5 Frequency Distance Filtering

As noted in [34], frequency distance displays great variation with increase in the number of uncertain positions in a string and can be effective to prune out dissimilar string pairs. We first obtain a simple lower bound on $fd(R, S)$ and then show how to quickly compute the upper bound for the same. For each character $c_i \in \Sigma$, let $f(S)_i^c$, $f(S)_i^t$ denote the minimum and maximum possible number of its occurrences in S respectively. For brevity, we drop the function notations and denote these occurrences as fS_i^c and fS_i^t. Note that fS_i^c also represents the number of occurrences of c_i in S with probability 1 and fS_i^t represents the number of certain and uncertain positions of c_i. Thus $fS_i^u = fS_i^t - fS_i^c$ gives the uncertain positions of c_i in S. fR_i^c, fR_i^u and fR_i^t are defined similarly. We observe that, if $fR_i^t < fS_i^c$, any possible world pw of $R \times S$, will have a frequency distance at least $(fS_i^c - fR_i^t)$. By generalizing this observation, we can obtain a lower bound on $fd(R, S)$ as summarized below.

Lemma 7.8. Let R and S be two strings from the same alphabet Σ, the we have $fd(R, S) \geq max\{pD, nD\}$, where

$$pD = \sum_{fS_i^t < fR_i^c} (fR_i^c - fS_i^t), nD = \sum_{fR_i^t < fS_i^c} (fS_i^c - fR_i^t)$$

Since the edit distance of a string pair is lower bounded by its frequency distance, we can prune out (R, S) if the minimum frequency distance obtained by above the lemma is more than the desired

edit threshold k. To obtain the upper bound on the probability of $fd(R, S)$ being at most k, we use the technique introduced in [34] that relies on the expected value of all possible frequency distances. Using such an expectation for positive and negative frequency distance ($E[pD]$, $E[nD]$), One-Sided Chebyshev Inequality and following the same analysis in [34], we obtain following theorem.

Theorem 7.9. Let R and S be two strings from the same alphabet Σ. Then we have,

$$\begin{aligned} Pr(ed(R,S) \leq k) &\leq Pr(fd(R,S) \leq k) \\ &\leq \frac{B^2}{B^2 + (A-k)^2} \end{aligned}$$

$$\text{where, } A = \frac{||R| - |S||}{2} + \frac{E[pD] + E[nD]}{2}$$

$$\begin{aligned} B^2 = \frac{(|R| - |S|)^2}{2} &+ \frac{(||R| - |S||)(E[pD] + E[nD])}{2} \\ &+ min(|R| \cdot E[nD], |S| \cdot E[pD]) - A^2 \end{aligned}$$

The main obstacle in using the above theorem is efficient computation of $E[pD] = \sum_{c_i} E(fR_i - fS_i)$, $E[nD] = \sum_{c_i} E(fS_i - fR_i)$. We focus on computing $E[nD]$ below as $E[pD]$ can be computed in a similar fashion. With frequency of c_i in S i.e., fS_i varying between fS_i^c and fS_i^t, let $Pr(fS_i = x)$ represents the probability that c_i appears exactly x times. Putting it an other way, $Pr(fS_i = x)$ represents the probability that c_i appears at exactly $(x - fS_i^c)$ uncertain positions from (fS_i^u) uncertain positions overall. This leads to a natural dynamic programming algorithm that can compute $Pr(fS_i = x)$ for all $x = fS_i^c, ..., fS_i^t$ by spending $O((fS_i^u)^2)$ time. Please refer to [34] more details. With the goal of efficiency in computing $E[nD]$, authors preprocess S and maintain these values in $O(fS_i^u)$ space. Without loss of generality, let $fR_i^c < fS_i^c \leq fR_i^t < fS_i^t$. Then by definition, $E[nD] = \sum_{c_i} E[nD_i]$ where,

$$E[nD_i] = \sum_{x=fR_i^c}^{fR_i^t} \sum_{\substack{y=max \\ (x+1, fS_i^c)}}^{fS_i^t} Pr(fR_i = x) Pr(fS_i = y)(y - x)$$

In the above equation, $Pr(fR_i = x)$ and $Pr(fS_i = y)$ can be computed in constant time using precomputed answers. Therefore, a naive way of computing $E[nD_i]$ will take $O(fS_i^u fR_i^u)$. Below we speed up this computation and achieve $min(fS_i^u, fR_i^u)$ time. We maintain the following probability distributions for each c_i of S. For $0 \leq x \leq fS_i^u$,

$$S1_i[x] = Pr(fS_i = fS_i^c + x)$$
$$S2_i[x] = \sum_{y=x}^{fS_i^u} Pr(fS_i = fS_i^c + y)$$
$$S3_i[x] = \sum_{y=x}^{fS_i^u} (y - x + 1) Pr(fS_i = fS_i^c + y)$$
$$S4_i[x] = \sum_{y=0}^{x} (x - y) Pr(fS_i = fS_i^c + y)$$

$S1_i$ is simply a probability distribution of c_i appearing at uncertain positions in range $[0, fS_i^u]$ (precomputed using dynamic programming). $S2_i$ maintains the probability that c_i appears at at-least x uncertain positions i.e., $S2_i[x] = Pr(fS_i \geq fS_i^c + x)$. $S3_i$ maintains the same summation with elements in the summation series scaled by $1, 2, ...$ Finally $S4_i$ takes the summation series for $Pr(fS_i \leq fS_i^c + x)$, scales it by $0, 1, ...$ in reverse direction and maintains the output at index x. The intuition behind maintaining the scaled summations is that, given a particular frequency z of fR_i, the expectation of its frequency distance with $fS_i \in [fS_i^c, fS_i^t]$ resembles the summation series for $S3_i[x]$ or $S4_i[x]$. All the above distributions can be computed in $O(fS_i^u)$ time and occupy the same $O(fS_i^u)$ storage. Similar probability distributions are also maintained for R. We achieve the speed up without hurting preprocessing time and at no additional storage cost. $E[nD_i]$ can now be computed as follows:

$$E[nD_i] = \sum_{x=fR_i^c}^{fS_i^c - 1} \sum_{y=fS_i^c}^{fS_i^t} (...) + \sum_{x=fS_i^c}^{fR_i^t} \sum_{y=x+1}^{fS_i^t} (...)$$
$$= nD_i^1 + nD_i^2$$

$$
\begin{aligned}
nD_i^1 &= \sum_{x=fR_i^c}^{fS_i^c-1} Pr(fR_i = x)(\sum_{y=fS_i^c}^{fS_i^t} Pr(fS_i = y)(y-x)) \\
&= \sum_{x=fR_i^c}^{fS_i^c-1} Pr(fR_i = x)(fS_i^c - x - 1) \sum_{y=fS_i^c}^{fS_i^t} Pr(fS_i = y) \\
&+ \sum_{x=fR_i^c}^{fS_i^c-1} Pr(fR_i = x) \sum_{y=fS_i^c}^{fS_i^t} Pr(fS_i = y)(y - fS_i^c + 1) \\
&= R4_i[fS_i^c - fR_i^c - 1] \times S2_i[0] \\
&+ (R2_i[0] - R2_i[fS_i^c - fR_i^c]) \times S3_i[0] \\
nD_i^2 &= \sum_{x=fS_i^c}^{fR_i^t} Pr(fR_i = x) \sum_{y=x+1}^{fS_i^t} Pr(fS_i = y)(y-x) \\
&= \sum_{x=fS_i^c}^{fS_i^t} R1_i[x - fR_i^c] \times S2_i[x - fS_i^c + 1]
\end{aligned}
$$

If the fraction of uncertain characters in the strings is θ, frequency filtering summarized in Theorem 7.9 can be applied in $O(\sigma\theta(|R|+|S|))$. Typical alphabet size being constant, the efficiency of applying frequency filtering depends on the degree of uncertainty and string lengths. Therefore, with increase in length of input strings, improvement from $|R| \times |S|$ to $|R|+|S|$ provides substantial reduction in the filtering time. While answering the similarity join query, we preprocess $R = S_i \in \mathcal{S}_l$ to compute the arrays for each character in alphabet Σ and maintain them as a part of our index. All candidate pairs passing the q-gram filtering are then subjected to frequency distance filtering for further refinement before moving onto next string $R = S_{i+1} \in \mathcal{S}_l$.

7.6 Verification

The goal of verification is to conclude whether strings in the candidate pair (R, S) that has survived the above filters, are indeed similar i.e., $Pr(ed(R, S) \leq k) > \tau$. A straightforward solution is to instantiate each possible world of $R \times S$ and add up the probabilities of possible worlds where possible instances of R, S are within edit threshold k. Before resorting to such expensive verification,

we make a last attempt to prune out a candidate pair, by extending the CDF bounds in [51]. If unsuccessful, we use the trie-based verification that exploits common prefixes shared by instances of an uncertain string.

7.6.1 Bound based on CDF

We briefly review the process in [51] and highlight the changes needed to compute the mentioned bounds correctly when both input strings are uncertain. We populate the matrix $|R| \times |S|$ using dynamic programming. In each cell $D = (x, y)$, we compute (at most) $k + 1$ pairs of values i.e., $\{(L[j], U[j])|0 \leq j \leq k\}$, where $L[j]$ and $U[j]$ are the lower and upper bounds of $Pr(ed(R[1..x], S[1..y]) \leq j)$ respectively. Then by checking the bounds in the cell $(|R|, |S|)$, we can accept or reject the candidate string pair (R, S), if possible. To fill in the DP table, consider a basic step of computing bounds of a cell $D = (x, y)$ from its neighboring cells - upper left: $D1 = (x-1, y-1)$, upper: $D2 = (x, y-1)$, and left: $D3 = (x-1, y)$. As noted in [51], when the $R[x]$ matches $S[y]$ (with probability $p_1 = \sum_{c_i} Pr(R[x] = c_i)Pr(S[y] = c_i)$), it is always optimal to take the distribution from the diagonal upper left neighbor. When $R[x]$ does not match $S[y]$ with probability $p_2 = 1 - p_1$, we use the relaxations suggested in [51]. Let $(argmin\ D_i)$ returns index i $(1 \leq i \leq 3)$ such that $L_{D_i}[0]$ is greatest; a tie is broken by selecting the greatest $L_{D_i}[1]$ and so on.

Theorem 7.10. At each cell $D = (x, y)$ of the DP table, $L[j] \leq Pr(ed(R[1..x], S[1..y]) \leq j) \leq U[j]$, where

$$L[j] = max(p_1 L_{D_1}[j],\ p_2 L_{(argmin\ D_i)}[j-1])$$

$$U[j] = min(1,\ p_1 U_{D_1}[j] + p_2 U_{D_1}[j-1] + \sum_{i=2}^{3} U_{D_i}[j-1])$$

Proof. We follow the analysis in [51] as follows. Consider a possible world $pw_{i,j}$ in which $r_i[x] = s_j[y]$. Let the distance values at cells D and D_i $(1 \leq i \leq 3)$ be v and v_i, respectively. Then we have $v = v_1$. This is because $v_2, v_3 \geq v_1 - 1$; thus, $v = min(v_1, v_2 + 1, v_3 + 1) = v_1$. Next, consider a possible world $pw_{i,j}$ in which $r_i[x] \neq s_j[y]$. Then, $v = min(v_i) + 1$. By using $(argmin\ D_i)$, we pick one fixed neighbor cell (i.e., the one that has a small distance value with the highest probability)

instead of accounting for all possible worlds in which $r_i[x] \neq s_j[y]$; and hence the true v value could be smaller than this one in some possible worlds. However, we observe that out of all possible worlds with distance v in the cell D, worlds with edit distance v in D_1 are not disjoint with worlds with distance $v-1$ for D_2. The same argument applies for worlds with $v-1$ as distance in D_3 as well. Therefore, we choose the maximum out of the two scenarios as our lower bound. For obtaining the upper bound, the case where $r_i[x]$ matches $s_j[y]$ remains the same. Possible world $pw_{i,j}$ with distance $v-1$ for D_2, can be extended by reading an addition character of R and we get distance v in cell D for all of them. Similarly, moving from distance $v-1$ in D_3 to distance v in D can be thought to be the case of inserting a character of S. Hence, we do not need to scale down the probability $U_{D_2}[v-1]$ as well $U_{D_3}[v-1]$ to obtain the upper bound for cell D. □

We note that the bounds summarized in the above theorem are different than the ones presented in [51], as they cannot be used directly for the current scenario[1]. Finally, the simple DP algorithm can be improved by computing $(L[j], U[j])$ only for those cells $D = (x, y)$ for which $|x - y| \leq k$, since $L[k] = U[k] = 0$ otherwise. Thus, we can apply the CDF bounds based filtering for a candidate pair (R, S) in $O(min(|R|, |S|)(k+1)max(k, \gamma))$ where γ is average number of alternatives of an uncertain character.

7.6.2 Trie-based Verification

Prefix-pruning has been a popular technique to expedite verification of a deterministic string pair (r, s) for edit threshold k. A naive approach for this verification would be to compute the dynamic programming matrix (DP) of size $|r| \times |s|$ such that cell (x, y) gives the edit distance between $r[1...x]$ and $s[1...y]$. Prefix-pruning observes that if all cells in row x i.e., $(x, *)$ do not meet threshold k, then the following rows can not have cells with edit distance k or less i.e., $DP[i > x, *] > k$. Even using such an early termination condition, verifying all-pairs (all possible of instance of $R \times S$) for a candidate pair (R, S) can be expensive. With the goal of avoiding naive all-pairs comparison, we propose trie-based verification. Let $\mathcal{T}_S$ be the trie of all possible instances of S and $\mathcal{T}_R$ be the same

[1]a) Lower bound violation: $r = ACC$, $S = A\{(C, 0.7), (G, 0.1), (T, 0.1)\}$ with $k = 1$. b) Upper bound violation: $r = DISC$, $S = DI\{(C, 0.4), (S, 0.5), (R = 0.1)\}$ with $k = 1$.

for string R. Let node u in $\mathcal{T}_S$ represents a string u (obtained by concatenating the edge labels from root to node u), then all possible instances of S with u as a prefix are leaves in the subtree rooted at u. We say a node $u \in \mathcal{T}_S$ is similar to node $v \in \mathcal{T}_R$ if $ed(u, v) \leq k$. Using prefix-pruning then we have following observation [42]:

- Given $u \in \mathcal{T}_S$, $v \in \mathcal{T}_R$: if u is not similar to any ancestor of v, and v is not similar to any ancestor of u, any possible instance s of S with prefix u can not be similar to a possible instance r of R with v as its prefix.

Using the technique in [78, 42], we can compute a set of similar nodes in $\mathcal{T}_R$ for each node $u \in \mathcal{T}_S$. Then, if $u = s_j$ is a leaf node, each node $v = r_i \in \mathcal{T}_R$ in its similar set that is also a leaf node, gives us a possible world $pw_{i,j}$ whose probability contributes to $Pr(ed(R, S) \leq k)$. However techniques in [42] implicitly assume both trie structures are available. Here we propose on-demand construction of trie which avoids all possible instances of S to be enumerated. Note that we still need to build the trie $\mathcal{T}_R$ completely. However its construction cost can be amortized as we build $\mathcal{T}_R$ once and use it for all candidate pairs $(R, *)$. As noted in [78], nodes in $\mathcal{T}_R$ that are similar to node $u \in \mathcal{T}_S$ can be computed efficiently only using such a similarity set already computed for its parent. This allows us to perform a (logical) depth first search on $\mathcal{T}_S$ and materialize the children of $u \in \mathcal{T}_S$ only if its similarity set is not empty. Figure 7.1 illustrates of this approach and reveals that on-demand trie construction can reduce the verification cost by avoiding instantiation and consequently comparison with a large fraction of possible worlds of S. In the figure, only the nodes linked with solid lines are explored and instantiated by the verification algorithm. Moreover, we do not display the similar node sets and the probabilities associated with trie nodes for simplicity.

7.7 Experiments

We have implemented the proposed indexing scheme and filtering techniques in C++. The experiments are performed on a 64 bit machine with an Intel Core i5 CPU 3.33GHz processor and 8GB RAM running Ubuntu. We consider the following algorithms for comparisons which use only a subset of the filtering mechanisms. Algorithm QFCT makes use of all the filtering schemes listed

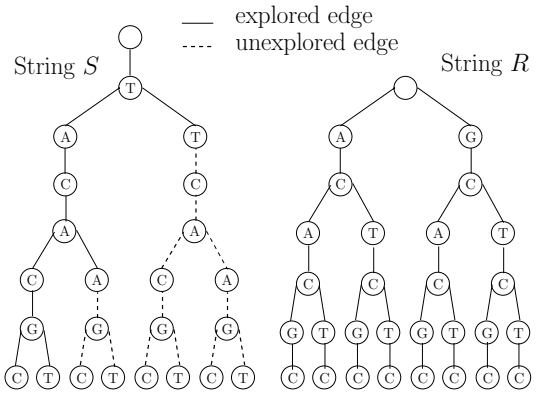

FIGURE 7.1. Trie-based verification example

in this article whereas QCT, QFT, FCT bypass frequency-distance filtering, filtering based on CDF bounds and q-gram filtering respectively.

Datasets: We use two synthetic datasets obtained from their real counterparts employing the technique used in [77, 34]. The first data source is the author names in dblp ($|\Sigma| = 27$). For each string s in the dblp dataset we first obtain a set $A(s)$ of strings that are within edit distance 4 to s. Then a character-level probabilistic string S for string s is generated such that, for a position i, the pdf of $S[i]$ is based on the normalized frequencies of the letters in the i-th position of all the strings in $A(s)$. The fraction of uncertain positions in a character-level probabilistic string i.e., θ is varied between 0.1 to 0.4 to generate strings with different degree of uncertainty. The string length distributions in this dataset follow approximately a normal distribution in the range of $[10, 35]$. For the second dataset we use a concatenated protein sequence of mouse and human ($|\Sigma| = 22$), and break it arbitrarily into shorter strings. Then uncertain strings are obtained by following the same procedure as that for the dblp data source. However, for this dataset we use slightly larger string lengths with less uncertainty i.e., string lengths roughly follow uniform distribution in the range $[20, 45]$ and θ ranges between 0.05 to 0.2. In both datasets, the average number of choices (γ) that each probabilistic character $S[i]$ may have is set to 5. The default values used for the dblp dataset are: the number of strings in collection $|\mathcal{S}|$ = 100K, average string length ≈ 19, θ = 0.2, k = 2, τ = 0.1, and q = 3. Similarly for protein dataset we use default setting with $|\mathcal{S}|$ = 100K, average string length $= 32$, θ = 0.1, k = 4, τ = 0.01, and q = 3.

7.7.1 Effectiveness vs Efficiency of Pruning

In this set of experiments, we compare the pruning ability of the filtering techniques and the overhead of applying them on both the datasets with $\theta = 0.2$, $k = 2$ and $\tau = 0.1$. Figure 7.2 shows the number of candidates remaining after applying each filtering scheme and reveals that CDF bounds provide the tightest filtering among the three. Effectiveness of the CDF follows from the fact that it uses upper as well as lower bounds to prune the strings. The upper bound obtained by q-gram filtering tends to be looser than the CDF as it depends on the partitioning based on q, whereas frequency distance based upper bound is sensitive to the length difference between two strings. However, the effectiveness of CDF comes at the cost of time. On the other hand, q-gram filtering is extremely fast and can still prune out a significant number of candidate pairs taking advantage of the indexing scheme. For the protein dataset, q-gram is close to CDF bounds in terms of effectiveness and is an order of magnitude faster than computing CDF bounds. Frequency distance filtering being dependent only on alphabet size and uncertain positions in the strings (against CDF's dependance on string length) can help to improve query performance by reducing the number of candidate pairs passed on to CDF for evaluation. Therefore, in the following experiments, algorithm variants use these filtering techniques in the increasing order of their overhead as suggested by their acronyms.

Figure 7.2 also reveals that applying q-gram filtering and the CDF bounds filtering takes longer for the protein dataset than for dblp data. Due to larger string length and fixed q, q-gram filtering needs to partition protein strings into a larger number of segments (i.e., m). Thus, there are more α_x probabilities to be computed and it takes longer to compute the desired upper bound in Theorem 7.7.

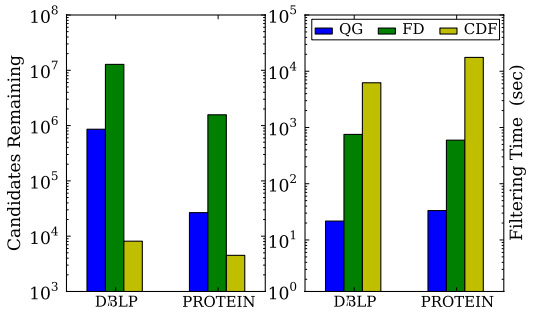

FIGURE 7.2. Effectiveness vs. efficiency

Similarly, computing CDF bounds needs to populate a dynamic programming matrix whose size depends on the string lengths. However, frequency distance filtering benefits from smaller alphabet set and lower degree of uncertainty in protein sequences and shows better performance for the protein data.

7.7.2 Effects of Data Size $|\mathcal{S}|$

Figure 7.3 shows the scalability of various algorithms on the dblp dataset, where we vary $|\mathcal{S}|$ from 50K to 500K. With computationally inexpensive q-gram filtering as the first step, algorithms QFCT, QFT and QCT achieve efficient filtering even for the larger datasets. For the exceptional case of the algorithm FCT, the filtering overhead increases almost quadratically with increase in the input size as both filtering techniques (frequency distance and CDF bounds) need to explicitly compare the query string R with all possible strings $S \in \mathcal{S}$ ($|S| \geq |R| - k$). Also, the filtering time required for QFT and QCT closely follows that for QFCT. This confirms the ability of q-gram filtering to significantly reduce the filtering overhead, and highlights the advantages offered by the proposed indexing scheme incorporating it.

Figure 7.3 also shows the time required for answering the join query for these algorithms. FCT, lacking efficient filtering (though effective), takes the longest to output its answers. However, the query time for QFT, despite using efficient q-gram filtering, shows a rapid increase. In contrast to this, the good scalable behavior of QFCT and QCT emphasizes the need for using tight filtering conditions based on the lower and upper bounds of CDF. In the absence of these, exponentially

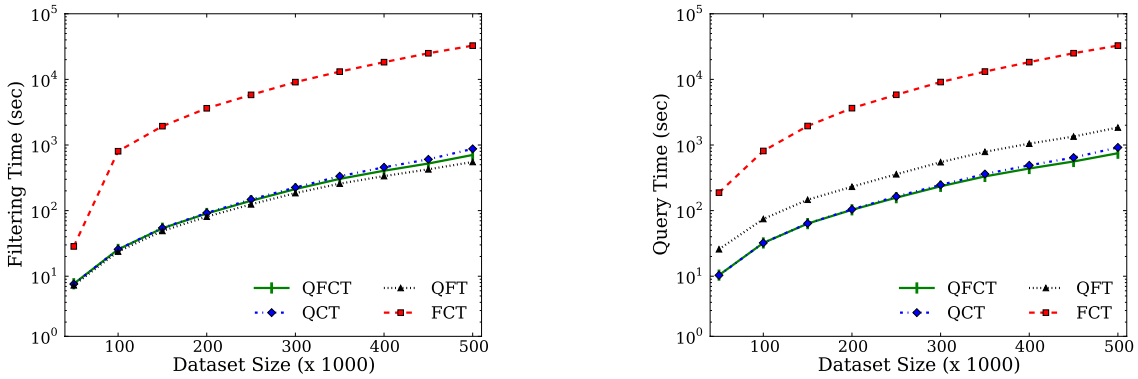

FIGURE 7.3. Effect of dataset size $|\mathcal{S}|$

more number of candidates need trie-based verification which results in quickly deteriorating query performance. Thus, a combination of q-gram filtering with CDF bounds in QFCT achieves the best of both worlds, allowing us to restrict the increase in both filtering time as well as the number of trie-based verifications. Though the number of outputs increased quadratically with data size, the increase in the number of false positives in the verification step of QFCT (i.e., the scenario where a candidate pair was not an output after verification) was found to be linear to the output size. An order of magnitude performance gain of QFCT over others seen in Figure 7.3 will be further extended for larger input collections. With algorithm QFT requiring a higher number of expensive verifications and QCT showing similar trends as that of QFCT, we use only the remaining two algorithms i.e., QFCT and FCT for the experiments to follow. We also append a character 'D' or 'P' to the algorithms acronym to distinguish between its query times on the dblp and protein datasets.

7.7.3 Effects of θ

An increase in the number of uncertain positions in the string has a detrimental effect on both algorithms QFCT and FCT as shown in Figure 7.4. This is due to the direct impact of θ on every step of the algorithm in answering join queries. Starting with the q-gram filtering, more uncertain positions for query string R imply more time required for populating the sets $q(r, x)$ as a preprocessing step, as well as for adding the string R to inverted indices after answering the query. Also the larger size of set $q(r, x)$ due to the increase in θ increases look up time in inverted indices

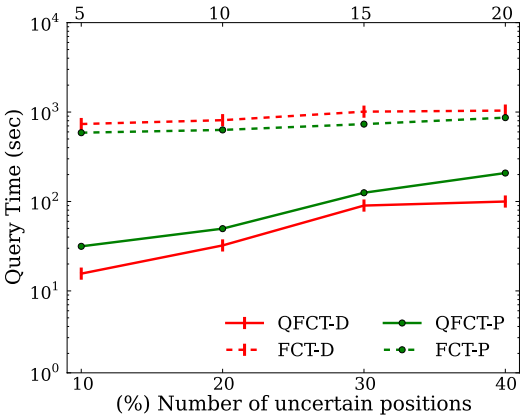

FIGURE 7.4. Effect of θ

and consequently increase the time required for computing α_x. Though size of a set $q(r, x)$ can increase exponentially with θ, its impact is limited due to the small fixed value of q. There is another subtle impact of θ on q-gram filtering. With more uncertain positions in query string R, more strings in the collection can be matched with substrings of R. We found this increase to be linear with $\approx 1.5\%$ of all join pairs evaluated by q-gram filtering for $\theta = 0.1$ to only $\approx 4\%$ evaluated for $\theta = 0.4$ on the dblp dataset. Thus, proposed q-gram filtering serves the purpose of efficient pruning even with the increased uncertainty.

The impact of θ on the computation of frequency distance and CDF bounds is more obvious. Computing the expected frequency distance of a character directly depends on the number of positions in input strings (R, S) where it appears probabilistically. Due to the probability computation of two positions matching in R and S ($R[x] = S[y]$), it takes longer to populate a dynamic programming matrix for CDF. Thus, the increase in filtering time of query algorithms is almost linear to θ. Finally, in the trie-based verification, more possible words need to be evaluated, increasing verification cost exponentially. In conclusion, the verification step is the worst affected among all due to large θ and is the primary contributor in increased time for answering join queries. We note that in most of the scenarios, algorithm QFCT takes longer to answer join queries for the protein data than for the dblp data because of the higher overhead of q-gram and CDF filtering, which we pointed out in Section 7.7.1. On the other hand, algorithm FCT performs better for the protein data by virtue of faster frequency filtering as seen earlier. This comparative behavior of QFCT and FCT is also evident in Figure 7.4.

7.7.4 Effects of τ

Figure 7.5 shows the results on the dblp and protein dataset for different values of τ from 0.001 to 0.4. Though the query times remain insensitive to τ for a large range, a gradual increase or decrease in probability threshold has a two fold effect on query algorithms. We analyze the scenario by looking at the number of candidate pairs pruned by CDF bounds either by accepting based on lower bound or rejecting based on upper bound. As τ increases, upper bound filter becomes more and more selective as it can reject more number of candidate pairs. On the contrary, filtering based

on lower bound looses its effectiveness with increased τ as it can not accept as many strings as it can for smaller values of τ. Thus the relative increase and decrease in number of candidate pairs pruned by CDF upper and lower bound respectively determines the overall effect of varying τ. When upper bound filter can not compensate for the loss in effectiveness of lower bound, more candidate pairs require trie-based verification resulting in higher query time. Such a scenario is evident in Figure 7.5 for protein data for τ ranging from 0.001 to 0.1.

τ has an interesting effect on q-gram filtering. Figure 7.5 shows the number of candidates pairs rejected by q-gram filtering in QFCT. It also shows the count of accepted candidates using CDF lower bound and rejected by CDF upper bound in QFCT. Note that q-gram filtering only uses the upper bound and Figure 7.5 shows the reduced effectiveness of CDF lower bound filtering. As τ increases, probabilistic pruning (Theorem 7.7) becomes more effective and prunes out a significant number of candidate pairs that satisfy the necessary condition for two strings to be similar as described in Lemma 7.6 (shown in Figure 7.5). In effect, q-gram filtering reduces the overhead of applying CDF bounds and to some extent compensates for the increased verification cost, if any. This effect can be seen by gradual decrease in the number of candidates rejected by CDF even though an increased number of candidates are pruned using the upper bound overall. Finally, for large τ, the q-gram filtering advantage coupled with reduced output size due to more selective τ results in improved query time.

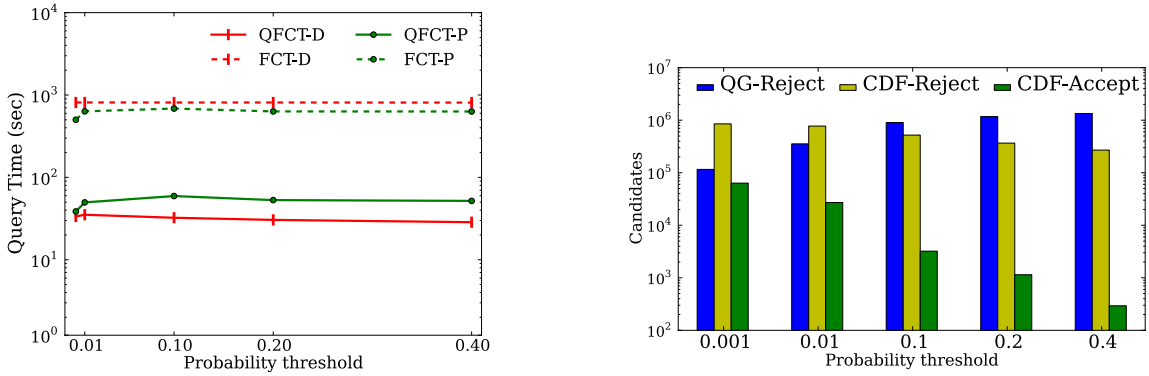

FIGURE 7.5. Effect of τ

7.7.5 Effects of k

Figure 7.6 shows the time required for answering a join query on the dblp dataset when k changes from 1 to 4 and for the protein dataset with $k = 2, 4, 6, 8$. With increased k we can expect more string pairs to satisfy an edit threshold and hence an increase in query time. As we loosen the edit threshold requirement, the effectiveness of q-gram filtering begins to deteriorate since the requirement for Lemma 7.6 can be met with string S having less number of its partitions being matched with substrings in R. Therefore, even with probabilistic pruning, many false candidates pairs are passed on to frequency distance and CDF filtering routines. Also, the number of candidates removed by the upper bound of frequency distance and CDF decreases with an increase in k. Though lower bound filtering in CDF can accept more candidates with an increase in k, this benefit is easily offset by loose upper bounds resulting in net increase in verification cost. With increased k, the time required for QFCT approaches that of FCT but still manages to save up to 35% of FCT's query cost.

7.7.6 Effects of q

In this set of experiments we try to investigate the effect of q-gram length on the efficiency and effectiveness of q-gram filtering using input collections with 100K strings. As pointed out earlier, q-gram filtering incurs more filtering overhead for higher string lengths with fixed q. We can hope to reduce this overhead by increasing q, however such an increase has side-effects on the space-time tradeoff of q-gram filtering. Even though we will have fewer partitions for each string

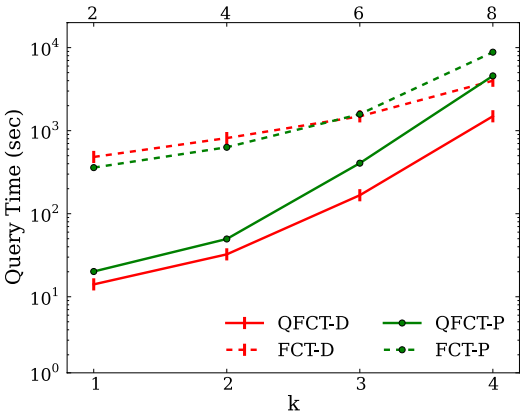

FIGURE 7.6. Effect of k

due to increased q, each segment now has more possible instances to be added to the inverted indices increasing the storage requirement as shown in Figure 7.7. The rate of increase is faster for the dblp dataset because of higher θ i.e., more uncertain positions and larger alphabet set. We note that we use peak memory usage as a measure that accounts for the indices maintained at any point during query answering based on the length of a string currently under consideration. Further, this also implies that query preprocessing that populates sets $q(r, x)$ needs more time offsetting the benefits of higher q to some extent. Figure 7.7 shows the improvement in the filtering time for q varying from 2 to 6. With size of $q(r, x)$ increasing exponentially with q, the improvement in filtering time achieved due to fewer segments also decreases exponentially.

For deterministic strings, increasing q makes it difficult for a segment of string s to match with substrings of query string r and implies potential improvement in pruning ability of q-gram filtering. For uncertain strings though, due to higher q, a segment may contain a larger number of uncertain positions. Hence there are more number of possible instances with increased chances for a segment to find a match in substrings of a query string. As a result, the effectiveness of q-gram filtering diminishes gradually for higher q as seen in Figure 7.7. We note that, though filtering time improves with q, time required for answering a join query shows uni-valley behavior as less effective filtering causes increased query time for higher q even with less filtering overhead. We found $q = 3$ or $q = 4$ offers the best combination of fast effective pruning with acceptable storage requirement. With peak memory usage of inverted indices less than the input data size itself for both $q = 3$ and 4, the space

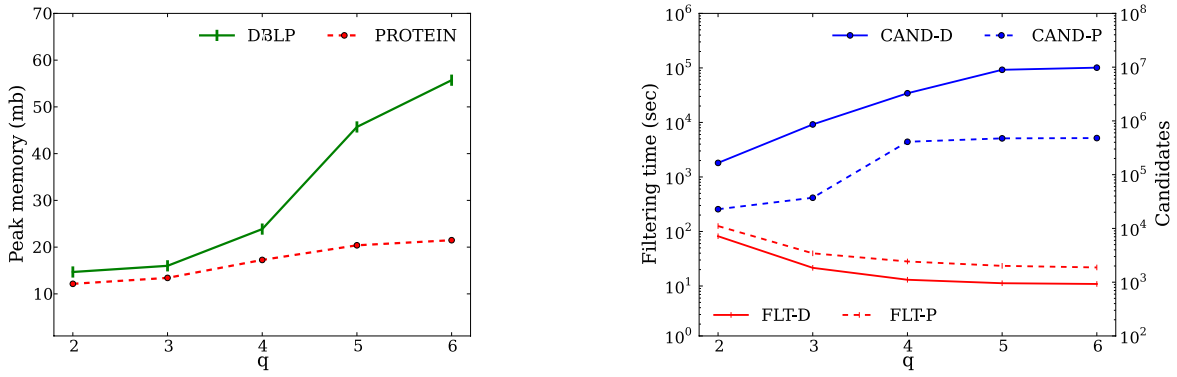

FIGURE 7.7. Effect of q

required for storing all indices as required for answering similarity search queries was found to be only ≈ 1.5 and ≈ 2 times the data size respectively.

7.7.7 Evaluating Trie-based Verification

We now analyze the performance benefits offered by the trie-based verification over a naive way of doing the same. Figure 7.8 shows the verification time required for answering join queries on the dblp and protein datasets with varying degree of uncertainty i.e., parameter θ. With an increase in the number of uncertain positions in the string, the number of possible worlds increases exponentially. This results in increased verification cost for both trie-based and naive verification. In naive verification, we need to enumerate possible worlds for each string in the dataset and also enumerate possible words for each of the candidate strings that may form a similar pair with it. In effect, we may enumerate all possible worlds for each string more than once. Additionally, given a candidate pair (R, S) it needs to compare every possible instance of R with that of S. In contrast, the trie-based verification enumerates all possible words for each string S only once and when it is selected as a candidate for some other string R in database, it enumerates and compares only those possible worlds which are highly likely to be similar to some instance of R. Thus the performance gains of trie-based verification increase with increasing θ as seen in Figure 7.8. We note that the cost of verification using trie-based approach also increases exponentially due to the requirement of having a complete trie in place for query string R. Moreover, trie-based verification can be more expensive than the naive method in scenarios where the majority of instances of $R \times S$ satisfy the edit threshold due to the overhead of building a trie and computing a set of similar nodes for each node in the trie. Though we obtained performance gains using trie-based verification on the protein data as well, they were less significant than for the dblp data due to higher string lengths, lower degree of uncertainty (θ) and smaller alphabet set.

7.7.8 Effects of String Length

In this final set of experiments, we test algorithms QFCT and FCT by varying the length of the probabilistic strings. For studying this effect, we use the 100K versions of the dblp and protein

datasets, and append each probabilistic string to itself for 0,1,2 or 3 times. To ensure that the verification step does not get excessively expensive, we limit the number of probabilistic characters in a probabilistic string to be at most 8. Clearly, the costs of both algorithms increase with longer strings as seen in Figure 7.9. In terms of filtering time, computation of q-gram filtering and CDF bounds takes longer as string lengths increase, as described earlier in Section 7.7.1. However, frequency distance filtering being dependent only on the number of uncertain character positions remains unaffected. This allows algorithm FCT to close the performance gap with QFCT for higher string lengths by virtue of efficient frequency distance filtering. Additionally, verification cost begins to dominate the query time with the increase in string lengths. We note that even trie-based verification needs to instantiate all possible worlds for each probabilistic string once while answering a join query. With each possible world enumeration taking more time, higher string length adversely affects the verification step. For fixed k, τ, and uncertain character positions, the number of output pairs decreases with increase in string length. Despite this, the query time increases because of the aforementioned reasons. We emphasize that the proposed filtering techniques maintain their effectiveness with varying lengths as the fraction of the candidate pairs that undergo verification and are accepted as output remains almost constant.

7.7.9 Comparison with EED

In this subsection, we qualitatively compare the join query algorithm in [77] against algorithms presented in this work:

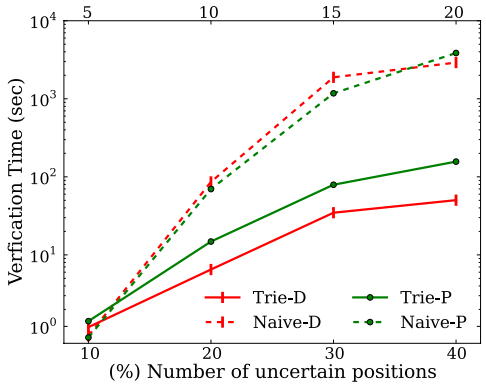

FIGURE 7.8. Trie-based verification

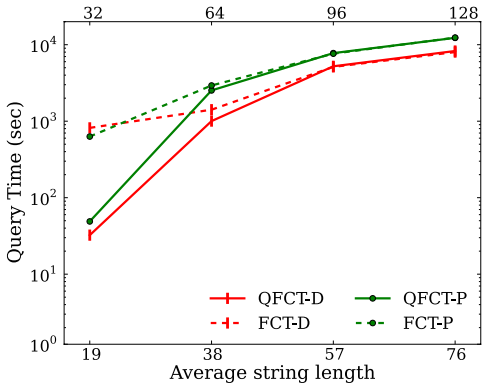

FIGURE 7.9. Effects of string length

1. We partition each string in the collection based on q whereas q-gram filtering in [77] makes use of overlapping q-grams. This allows us to significantly reduce the space required for storing all q-grams ($\approx 5 \times datasize$ as reported in [77] against our index of twice the input data size).
2. q-gram filtering presented in [77] requires each probabilistic string pair to be evaluated during query execution tasks like computation of frequency distance, CDF bounds computation. Algorithm QFCT employes indexing that incorporates q-gram filtering before applying expensive filters. Therefore, we can expect QFCT to offer benefits over the query algorithm in [77] similar to its advantages over algorithm FCT seen in Figure 7.3.
3. Computing the exact eed between two probabilistic strings requires all possible worlds for two strings to be instantiated in the same way as a naive verification method discussed in Section 7.7.7. On the other hand trie-based verification allows us to determine the similarity of a string pair efficiently (refer to Figure 7.8).

7.8 Summary

In this chapter, we study the largely unexplored problem of answering similarity join queries on uncertain strings. We propose a novel q-gram filtering technique that integrates probabilistic pruning and extends frequency distance and CDF based filtering techniques. In future work, we plan to investigate tighter filtering conditions and improvements to the trie-based verification algorithm.

Chapter 8
Conclusions and Future Work

We have presented efficient data structures for several problems with applications in database and information retrieval systems dealing with structured/unstructured and precise/uncertain data. Most of these structures have been developed keeping RAM model in mind. Exponential growth of digital data in recent years has necessitated development of disk-resident indexing solutions. Moreover, limited scope for increasing processing power of uniprocessors has fueled interest in distributed construction, storage, and query processing of indexes. We conclude with interesting variations/extensions of the problems studied in this dissertation that need to explored in external memory and (or) distributed computing model.

- As pointed out in Chapter 3, top-k join problem is closely related to computing skylines. An extension to the classical skyline problem called range skyline asks for skyline points only among the points that fall within the query region [82]. It would be interesting to extend this problem for categorical data for three-sided query region of the form $[a, b] \times [\tau, +\infty]$ analogous to the work in Chapter 5.

- In chapter 4 we have investigated top-k document retrieval for a given document collection in internal memory. The proposed practical framework occupies close to twice the data size. This limits its usage for large collections which can not fit in internal memory. Implementing a disk-resident index for ranked document retrieval based on theoretical results in Chapter 5 will be helpful for numerous bioinformatics applications dealing with dna or protein sequences.

- Suffix tree is a widely used indexing data structure for many of the sequence based problems, such as pattern matching (Chapter 4, 5), finding the substrings etc. Due to the significance of suffix tree, many construction algorithms have been proposed [128, 95] for the same. When the string and the resulting suffix tree are too large to fit into the main memory,

these construction algorithms become very inefficient. Disk-based suffix tree construction methods have been proposed in the recent past [115, 12, 52] that work efficiently with very long strings. These techniques partition the suffix tree and aim to build each partition independently and sequentially within the available primary memory. Exploiting popular distributed computing models such as MapReduce for parallel suffix tree construction so as to improve the construction time as well as scalability of existing methods remains challenging.

- Techniques proposed in Chapter 7 can be used to answer similarity search queries over a collection of uncertain strings. Here, given a query string as input we need to output all the strings in the collection that are within the required edit distance threshold with probability higher than the input threshold [34]. However, with such a threshold being domain dependent, estimating the same to achieve reasonable output size is a non-trivial task. For deterministic strings, top-k query eliminates the need for such an estimation by requiring k most similar strings to be reported [37]. Extending top-k semantics to the uncertain strings, so as to only retrieve the k strings that have the highest probability of being within the desired edit distance threshold, is an important open problem.

Bibliography

[1] Peyman Afshani. On dominance reporting in 3d. In *Proceedings of European Symposium on Algorithms*, pages 41–51, 2008.

[2] A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1998.

[3] R. Akbarinia, I.F. Ilyas, M.T. Özsu, and P. Valduriez. Jtop algorithms for top-k join queries. In *Technical report*, 2008.

[4] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Best position algorithms for top-k queries. In *Proceedings of International Conference on Very Large Data Bases*, pages 495–506, 2007.

[5] Amihood Amir, Eran Chencinski, Costas S. Iliopoulos, Tsvi Kopelowitz, and Hui Zhang. Property matching and weighted matching. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 188–199, 2006.

[6] V. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of Special Interest Group on Information Retrieval*, pages 372–379, 2006.

[7] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *Proceedings of International Conference on Very Large Data Bases*, pages 918–929, 2006.

[8] Lars Arge, Vasilis Samoladas, and Jeffrey Scott Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings of Symposium on Principles of Database Systems*, pages 346–357, 1999.

[9] R. Baeza-Yates and B. Ribeiro-Neto. Modern information retrieval. 1999.

[10] Dirk Bahle, Hugh E. Williams, and Justin Zobel. Compaction techniques for nextword indexes. In *Proceedings of International Symposium on String Processing and Information Retrieval*, 2001.

[11] Dirk Bahle, Hugh E. Williams, and Justin Zobel. Optimised phrase querying and browsing of large text databases. In *ACSC*, pages 11–19, 2001.

[12] Marina Barsky, Ulrike Stege, Alex Thomo, and Chris Upton. Suffix trees for very large genomic sequences. In *Conference on Information and Knowledge Management*, pages 1417–1420, 2009.

[13] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. In *Proceedings of International Conference on Very Large Data Bases*, pages 475–486, 2006.

[14] M. A. Bender and M. Farach-Colton. The Level Ancestor Problem Simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

[15] Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.

[16] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.

[17] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[18] Panayiotis Bozanis, Nectarios Kitsios, Christos Makris, and Athanasios K. Tsakalidis. New upper bounds for generalized intersection searching problems. In *Proceedings of International Colloquium on Automata, Languages and Programming*, pages 464–474, 1995.

[19] Gerth Stølting Brodal, Rolf Fagerberg, Mark Greve, and Alejandro López-Ortiz. Online sorted range reporting. In *Proceedings of International Symposium on Algorithms and Computation*, pages 173–182, 2009.

[20] Nicolas Bruno, Luis Gravano, and Amélie Marian. Evaluating top-k queries over web-accessible databases. In *Proceedings of International Conference on Data Engineering*, 2002.

[21] Rodrigo Cánovas and Gonzalo Navarro. Practical compressed suffix trees. In *SEA*, pages 94–105, 2010.

[22] Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In *PODC*, pages 206–215, 2004.

[23] Amit Chakrabarti, Graham Cormode, and Andrew McGregor. Robust lower bounds for communication and stream computation. In *Proceedings of Symposium on Theory of Computing*, pages 641–650, 2008.

[24] Amit Chakrabarti, T. S. Jayram, and Mihai Patrascu. Tight lower bounds for selection in randomly ordered streams. In *Proceedings of Symposium on Discrete Algorithms*, pages 720–729, 2008.

[25] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *Proceedings of Special Interest Group on Management of Data*, pages 346–357, 2002.

[26] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of Special Interest Group on Management of Data*, pages 313–324, 2003.

[27] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of International Conference on Data Engineering*, page 5, 2006.

[28] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.

[29] Bernard Chazelle and Herbert Edelsbrunner. Linear space data structures for two types of range search. *Discrete & Computational Geometry*, 2:113–126, 1987.

[30] Jiang Chen and Ke Yi. Dynamic structures for top-k queries on uncertain data. In *Proceedings of International Symposium on Algorithms and Computation*, pages 427–438, 2007.

[31] Reynold Cheng, Dmitri V. Kalashnikov, and Sunil Prabhakar. Evaluating probabilistic queries over imprecise data. In *Proceedings of Special Interest Group on Management of Data*, pages 551–562, 2003.

[32] Graham Cormode, Feifei Li, and Ke Yi. Semantics of ranking queries for probabilistic data and expected ranks. In *Proceedings of International Conference on Data Engineering*, pages 305–316, 2009.

[33] J. Shane Culpepper, Gonzalo Navarro, Simon J. Puglisi, and Andrew Turpin. Top-k ranked document search in general text databases. In *Proceedings of European Symposium on Algorithms*, pages 194–205, 2010.

[34] Dongbo Dai, Jiang Xie, Huiran Zhang, and Jiaqi Dong. Efficient range queries over uncertain strings. In *Proceedings of International Conference on Scientific and Statistical Database Management*, pages 75–95, 2012.

[35] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *Proceedings of International Conference on Very Large Data Bases*, pages 864–875, 2004.

[36] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. Answering top-k queries using views. In *Proceedings of International Conference on Very Large Data Bases*, pages 451–462, 2006.

[37] Dong Deng, Guoliang Li, and Jianhua Feng. Top-k string similarity search with edit-distance constraints. In *Proceedings of International Conference on Data Engineering*, 2013.

[38] Amol Deshpande, Carlos Guestrin, Samuel Madden, Joseph M. Hellerstein, and Wei Hong. Model-driven data acquisition in sensor networks. In *Proceedings of International Conference on Very Large Data Bases*, pages 588–599, 2004.

[39] Ronald Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999.

[40] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing top k lists. In *Proceedings of Symposium on Discrete Algorithms*, pages 28–36, 2003.

[41] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of Symposium on Principles of Database Systems*, 2001.

[42] Jianhua Feng, Jiannan Wang, and Guoliang Li. Trie-join: a trie-based method for efficient string similarity joins. *VLDB J.*, 21(4):437–461, 2012.

[43] P. Ferragina and R. Grossi. The String B-tree: A New Data Structure for String Searching in External Memory and Its Application. *Journal of the ACM*, 46(2):236–280, 1999.

[44] Jonathan Finger and Neoklis Polyzotis. Robust and efficient algorithms for rank join evaluation. In *Proceedings of Special Interest Group on Management of Data*, pages 415–428, 2009.

[45] J. Fischer and V. Heun. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *ESCAPE*, pages 459–470, 2007.

[46] Johannes Fischer, Volker Heun, and Horst Martin Stühler. Practical entropy-bounded schemes for o(1)-range minimum queries. In *Proceedings of Data Compression Conference*, pages 272–281, 2008.

[47] Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. In *Theoretical Computer Science*, pages 5354–5364, 2009.

[48] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.

[49] T. Gagie, G. Navarro, and S. J. Puglisi. Colored Range Queries and Document Retrieval. In *Proceedings of International Symposium on String Processing and Information Retrieval*, pages 67–81, 2010.

[50] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model, and algorithms. In *Proceedings of International Conference on Very Large Data Bases*, pages 371–380, 2001.

[51] Tingjian Ge and Zheng Li. Approximate substring matching over uncertain strings. *PVLDB*, 4(11):772–782, 2011.

[52] Amol Ghoting and Konstantin Makarychev. Serial and parallel methods for i/o efficient suffix tree construction. In *Proceedings of Special Interest Group on Management of Data*, pages 827–840, 2009.

[53] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of International Conference on Very Large Data Bases*, 2001.

[54] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[55] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of Symposium on Discrete Algorithms*, pages 841–850, 2003.

[56] Sudipto Guha and Andrew McGregor. Approximate quantiles and the order of the stream. In *Proceedings of Symposium on Principles of Database Systems*, pages 273–279, 2006.

[57] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In *Proceedings of International Conference on Very Large Data Bases*, pages 419–428, 2000.

[58] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, pages 622–628, 2001.

[59] Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theor. Comput. Sci.*, 387(3):313–331, 2007.

[60] Prosenjit Gupta, Ravi Janardan, and Michiel H. M. Smid. Further results on generalized intersection searching problems: counting, reporting, and dynamization. *Journal of Algorithms*, 19(2):282–317, 1995.

[61] Alon Y. Halevy, Anand Rajaraman, and Joann J. Ordille. Data integration: The teenage years. In *Proceedings of International Conference on Very Large Data Bases*, pages 9–16, 2006.

[62] W. K. Hon, M. Patil, R. Shah, and S. B. Wu. Efficient Index for Retrieving Top-k Most Frequent Documents. *Journal on Discrete Algorithms*, 8(4):402–417, 2010.

[63] W. K. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. String Retrieval for Multi-pattern Queries. In *Proceedings of International Symposium on String Processing and Information Retrieval*, pages 55–66, 2010.

[64] Wing Kai Hon, Manish Patil, Rahul Shah, and Sharma V. Thankachan. Compressed property suffix trees. In *Proceedings of Data Compression Conference*, pages 123–132, 2011.

[65] Wing-Kai Hon, Rahul Shah, and Sharma V. Thankachan. Towards an optimal space-and-query-time index for top-k document retrieval. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 173–184, 2012.

[66] Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Space-efficient framework for top-k string retrieval problems. In *Proceedings of Symposium on Foundations of Computer Science*, pages 713–722, 2009.

[67] Ming Hua, Jian Pei, Wenjie Zhang, and Xuemin Lin. Ranking queries on uncertain data: a probabilistic threshold approach. In *Proceedings of Special Interest Group on Management of Data*, pages 673–686, 2008.

[68] Jiewen Huang, Lyublena Antova, Christoph Koch, and Dan Olteanu. Maybms: a probabilistic database management system. In *Proceedings of Special Interest Group on Management of Data*, pages 1071–1074, 2009.

[69] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal of Computing*, 1(1):31–39, 1972.

[70] Costas S. Iliopoulos, Christos Makris, Yannis Panagis, Katerina Perdikuri, Evangelos Theodoridis, and Athanasios Tsakalidis. The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. *Fundam. Inf.*, 71:259–277, February 2006.

[71] Costas S. Iliopoulos, Katerina Perdikuri, Evangelos Theodoridis, Athanasios K. Tsakalidis, and Kostas Tsichlas. Algorithms for extracting motifs from biological weighted sequences. *J. Discrete Algorithms*, 5(2):229–242, 2007.

[72] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Joining ranked inputs in practice. In *Proceedings of International Conference on Very Large Data Bases*, pages 950–961, 2002.

[73] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. In *Proceedings of International Conference on Very Large Data Bases*, pages 754–765, 2003.

[74] Ihab F. Ilyas, Walid G. Aref, Ahmed K. Elmagarmid, Hicham G. Elmongui, Rahul Shah, and Jeffrey Scott Vitter. Adaptive rank-aware query optimization in relational databases. *ACM Trans. Database Syst.*, 31(4):1257–1304, 2006.

[75] Ravi Janardan and Mario A. Lopez. Generalized intersection searching problems. *International Journal of Computational Geometry and Applications*, 3(1):39–69, 1993.

[76] J. Jestes, G. Cormode, F. Li, and K. Yi. Semantics of ranking queries for probabilistic data. *Knowledge and Data Engineering, IEEE Transactions on*, PP(99):1, 2010.

[77] Jeffrey Jestes, Feifei Li, Zhepeng Yan, and Ke Yi. Probabilistic string similarity joins. In *Proceedings of Special Interest Group on Management of Data*, pages 327–338, 2010.

[78] Shengyue Ji, Guoliang Li, Chen Li, and Jianhua Feng. Efficient interactive fuzzy keyword search. In *Proceedings of International Conference on World Wide Web*, pages 371–380, 2009.

[79] Cheqing Jin, Ke Yi, Lei Chen 0002, Jeffrey Xu Yu, and Xuemin Lin. Sliding-window top-k queries on uncertain streams. *PVLDB*, 1(1):301–312, 2008.

[80] Tamer Kahveci and Ambuj K. Singh. Efficient index structures for string databases. In *Proceedings of International Conference on Very Large Data Bases*, pages 351–360, 2001.

[81] Marek Karpinski and Yakov Nekrich. Top-k color queries for document retrieval. In *Proceedings of Symposium on Discrete Algorithms*, pages 401–411, 2011.

[82] Casper Kejlberg-Rasmussen, Yufei Tao, Konstantinos Tsakalidis, Kostas Tsichlas, and Jeonghun Yoon. I/o-efficient planar range skyline and attrition priority queues. In *Proceedings of Symposium on Principles of Database Systems*, pages 103–114, 2013.

[83] Christoph Koch and Dan Olteanu. Conditioning probabilistic databases. *Proc. VLDB Endow.*, 1:313–325, August 2008.

[84] Nick Koudas, Sunita Sarawagi, and Divesh Srivastava. Record linkage: similarity measures and algorithms. In *Proceedings of Special Interest Group on Management of Data*, pages 802–803, 2006.

[85] Kasper Green Larsen and Rasmus Pagh. I/o-efficient data structures for colored range and prefix reporting. In *Proceedings of Symposium on Discrete Algorithms*, pages 583–592, 2012.

[86] Kasper Green Larsen and Freek van Walderveen. Near-optimal range reporting structures for categorical data. In *Proceedings of Symposium on Discrete Algorithms*, pages 256–276, 2013.

[87] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Supporting ad-hoc ranking aggregates. In *Proceedings of Special Interest Group on Management of Data*, pages 61–72, 2006.

[88] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. Ranksql: Query algebra and optimization for relational top-k queries. In *Proceedings of Special Interest Group on Management of Data*, pages 131–142, 2005.

[89] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[90] Jian Li, Barna Saha, and Amol Deshpande. A unified approach to ranking in probabilistic databases. *PVLDB*, 2(1), 2009.

[91] Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3), 2008.

[92] Christos Makris and Athanasios K. Tsakalidis. Algorithms for three-dimensional dominance searching in linear space. *Inf. Process. Lett.*, 66(6):277–283, 1998.

[93] Nikos Mamoulis, Kit Hung Cheng, Man Lung Yiu, and David W. Cheung. Efficient aggregation of ranked inputs. In *Proceedings of International Conference on Data Engineering*, page 72, 2006.

[94] Yossi Matias, S. Muthukrishnan, Süleyman Cenk Sahinalp, and Jacob Ziv. Augmenting suffix trees, with applications. In *Proceedings of European Symposium on Algorithms*, pages 67–78, 1998.

[95] E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[96] Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. Klee: A framework for distributed top-k query algorithms. In *Proceedings of International Conference on Very Large Data Bases*, pages 637–648, 2005.

[97] J. I. Munro. Tables. In *Proceedings of Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42, 1996.

[98] J. I. Munro, V. Raman, and S. S. Rao. Space Efficient Suffix Trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[99] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of Symposium on Discrete Algorithms*, pages 657–666, 2002.

[100] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In *Proceedings of International Conference on Very Large Data Bases*, pages 281–290, 2001.

[101] Gonzalo Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. In *CoRR abs/304.6023*, 2013.

[102] Gonzalo Navarro and Yakov Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012)*, pages 1066–1077, 2012.

[103] Gonzalo Navarro and Simon J. Puglisi. Dual-sorted inverted lists. In *Proceedings of International Symposium on String Processing and Information Retrieval*, pages 309–321, 2010.

[104] Yakov Nekrich. Space-efficient range reporting for categorical data. In *Proceedings of Symposium on Principles of Database Systems*, pages 113–120, 2012.

[105] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proceedings of International Conference on Data Engineering*, pages 22–29, 1999.

[106] Enno Ohlebusch, Johannes Fischer, and Simon Gog. Cst++. In *Proceedings of International Symposium on String Processing and Information Retrieval*, pages 322–333, 2010.

[107] Enno Ohlebusch and Simon Gog. A compressed enhanced suffix array supporting fast string matching. In *Proceedings of International Symposium on String Processing and Information Retrieval*, pages 51–62, 2009.

[108] Manish Patil, Xuanting Cai, Sharma V. Thankachan, Rahul Shah, Seung-Jong Park, and David Foltz. Approximate string matching by position restricted alignment. In *EDBT/ICDT Workshops*, pages 384–391, 2013.

[109] Manish Patil and Rahul Shah. Similarity joins for uncertain strings. In *Proceedings of Special Interest Group on Management of Data*, pages 1471–1482, 2014.

[110] Manish Patil, Rahul Shah, and Sharma V. Thankachan. A truly dynamic data structure for top-k queries on uncertain data. In *Proceedings of International Conference on Scientific and Statistical Database Management*, pages 91–108, 2011.

[111] Manish Patil, Rahul Shah, and Sharma V. Thankachan. Top-k join queries: overcoming the curse of anti-correlation. In *IDEAS*, pages 76–85, 2013.

[112] Manish Patil, Sharma V. Thankachan, Rahul Shah, Wing-Kai Hon, Jeffrey Scott Vitter, and Sabrina Chandrasekaran. Inverted indexes for phrases and strings. In *Proceedings of Special Interest Group on Information Retrieval*, pages 555–564, 2011.

[113] Manish Patil, Sharma V. Thankachan, Rahul Shah, Yakov Nekrich, and Jeffrey Scott Vitter. Categorical range maxima queries. In *Proceedings of Symposium on Principles of Database Systems*, 2014.

[114] M. Persin, J. Zobel, and R. S. Davis. Filtered document retrieval with frequency-sorted indexes. In *Journal of the American Society for Information Science*, volume 47, pages 749–764, 1996.

[115] Benjarath Phoophakdee and Mohammed J. Zaki. Genome-scale disk-based suffix tree indexing. In *SIGMOD Conference*, pages 833–844, 2007.

[116] R. Raman, V. Raman, and S. S. Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

[117] L. Russo, G. Navarro, and A. Oliveira. Fully-Compressed Suffix Trees. In *Proceedings of Latin American Theoretical Informatics Symposium*, pages 362–373, 2008.

[118] K. Sadakane. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, pages 589–607, 2007.

[119] Kunihiko Sadakane. Space-efficient data structures for flexible text retrieval systems. In *Proceedings of International Symposium on Algorithms and Computation*, pages 14–24, 2002.

[120] Walter J. Savitch and Michael J. Stimson. Time bounded random access machines with parallel processing. *J. ACM*, 26(1):103–118, 1979.

[121] Prithviraj Sen, Amol Deshpande, and Lise Getoor. Prdb: managing and exploiting rich correlations in probabilistic databases. *VLDB J.*, 18(5):1065–1090, 2009.

[122] R. Shah, C. Sheng, S. V. Thankachan, and J. S. Vitter. Top-k document retrieval in external memory. In *Proceedings of European Symposium on Algorithms*, 2013.

[123] Jop F. Sibeyn. External selection. In *Proceedings of the 16th annual conference on Theoretical aspects of computer science*, STACS'99, pages 291–301, 1999.

[124] Sarvjeet Singh, Chris Mayfield, Sagar Mittal, Sunil Prabhakar, Susanne E. Hambrusch, and Rahul Shah. Orion 2.0: native support for uncertain data. In *Proceedings of Special Interest Group on Management of Data*, pages 1239–1242, 2008.

[125] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen-Chuan Chang. Top-k query processing in uncertain databases. In *Proceedings of International Conference on Data Engineering*, pages 896–905, 2007.

[126] Yufei Tao, Vagelis Hristidis, Dimitris Papadias, and Yannis Papakonstantinou. Branch-and-bound processing of ranked queries. *Inf. Syst.*, 32(3):424–445, 2007.

[127] Panayiotis Tsaparas, Themistoklis Palpanas, Yannis Kotidis, Nick Koudas, and Divesh Srivastava. Ranked join indices. In *Proceedings of International Conference on Data Engineering*, 2003.

[128] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[129] N. Valimaki and V. Makinen. Space-Efficient Algorithms for Document Retrieval. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 205–215, 2007.

[130] Niko Välimäki, Veli Mäkinen, Wolfgang Gerlach, and Kashyap Dixit. Engineering a compressed suffix tree implementation. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[131] Darren Erik Vengroff and Jeffrey Scott Vitter. Efficient 3-d range searching in external memory. In *Proceedings of Symposium on Theory of Computing*, pages 192–201, 1996.

[132] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[133] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.

[134] Hugh E. Williams, Justin Zobel, and Phil Anderson. What's next? index structures for efficient phrase querying. In *Australasian Database Conference*, pages 141–152, 1999.

[135] Hugh E. Williams, Justin Zobel, and Dirk Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, 2004.

[136] Chuan Xiao, Wei Wang, and Xuemin Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[137] Dong Xin, Jiawei Han, and Kevin Chen-Chuan Chang. Progressive and selective merge: computing top-k with ad-hoc ranking functions. In *Proceedings of Special Interest Group on Management of Data*, pages 103–114, 2007.

[138] Ke Yi, Feifei Li, George Kollios, and Divesh Srivastava. Efficient processing of top-k queries in uncertain databases. In *Proceedings of International Conference on Data Engineering*, pages 1406–1408, 2008.

[139] Hui Zhang, Qing Guo, and Costas S. Iliopoulos. An algorithmic framework for motif discovery problems in weighted sequences. In *CIAC*, pages 335–346, 2010.

[140] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

Vita

Manish Madhukar Patil was born in Mumbai, India, in 1982. He obtained his bachelor's degree in Computer Engineering in 2003 from Datta Meghe College of Engineering, University of Mumbai. During his doctoral studies at Louisiana State University, he has co-authored 10 refereed conference papers and 4 journal publications (published or accepted for publication by August 2014). His research interest falls in the area of algorithms and data structures with applications in database systems, information retrieval, and bioinformatics.